aws

Developer Guide

Managed Integrations for AWS IoT Device Management



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

Managed Integrations for AWS IoT Device Management: Developer Guide

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

What is managed integrations for AWS IoT Device Management 1
Supported Regions 1
Are you a first-time Managed Integrations user? 1
Managed Integrations overview 1
Managed Integrations terminology 2
General Managed Integrations terminology 2
Cloud-to-cloud terminology 3
Data model terminology 3
Set up Managed Integrations 5
Sign up for an AWS account 5
Get started 6
Device types 6
Configure encryption key 7
Onboarding techniques 7
Direct-connected device onboarding 8
Hub onboarding 8
Hub-connected device onboarding 8
Cloud-to-cloud device onboarding 8
Device provisioning 9
Manage device lifecycle and profiles 11
Device 11
Device profile 12
Data models 13
Managed Integrations data model 13
AWS implementation of the Matter data model 15
Data model schemas 16
Capability schema 17
Type definition schema 17
Schema for capability definitions 18
Schema for type definitions 35
Building and using type definitions in capability schema documents 40
Device commands and events 54
Device commands 54
Device Events 56

Tag resources 58
Tag basics 58
Tag restrictions and limitations 59
Tag with IAM policies 59
Managed Integrations notifications 63
Set up Amazon Kinesis for notifications 63
Step 1: Create an Amazon Kinesis data stream 63
Step 2: Create a permissions policy 63
Step 3: Navigate to the IAM dashboard and select Roles 64
Step 4: Use a Custom trust policy 64
Step 5: Apply your permissions policy 65
Step 6: Enter a role name 65
Set up Managed Integrations notifications 66
Step 1: Give user permissions to call the CreateDestination API 66
Step 2: Call the CreateDestination API 67
Step 3: Call the CreateNotificationConfiguration API 67
Event types monitored with Managed Integrations 68
Cloud-to-Cloud (C2C) connectors 77
What is a cloud-to-cloud (C2C) connector? 77
Connector catalog 77
AWS Lambda functions as C2C connectors 78
Managed Integrations connector workflow 78
Guidelines in using a C2C (cloud-to-cloud) connector 79
Build a C2C (Cloud-to-Cloud) connector 79
Prerequisites 80
Required permissions 81
C2C connector requirements 81
Authorization 83
Implement C2C connector interface operations 96
Invoke your C2C connector 126
Add permissions to your IAM Role 127
Manually test your C2C connector 128
Use a C2C (Cloud-to-Cloud) connector 128
General/Custom Authorization requirements 145
Hub SDK 148
Hub SDK architecture 148

Device onboarding ... 149
Device onboarding components ... 149
Device onboarding flows ... 150
Device control ... 155
Device control flows ... 156
SDK components ... 157
Install and validate the managed integrations Hub SDK ... 158
Install the SDK using AWS IoT Greengrass ... 158
Deploy the Hub SDK with a script ... 161
Deploy Hub SDK with systemd ... 164
Onboard your hubs ... 167
Hub onboarding subsystem ... 168
Setup for onboarding ... 169
Onboard devices and operate them in hub ... 178
Simple setup to onboard and operate devices ... 178
User guided setup to onboard and operate devices ... 185
Capability rediscovery ... 194
WiFi Simple Setup to onboard and operate devices ... 199
Custom certificate handler ... 223
API definition and components ... 224
Example build ... 226
Usage ... 230
Custom protocol plugin ... 231
Matter Plugin ... 232
What is Matter Plugin ... 233
Build Matter Plugin ... 235
Quickstart Setup ... 236
Raspberry Pi Steps ... 238
Supported Device Types ... 242
Additional Clusters ... 242
Integration Options ... 243
Hub SDK client ... 245
Get your managed integrations Hub SDK ... 245
About the Hub SDK toolkit ... 245
Create your custom application with the Hub SDK client ... 246
Running your custom application ... 248

Hub SDK client API ... 249
Data types ... 253
Hub control ... 255
Prerequisites ... 255
End device SDK components ... 256
Integrate with the End device SDK ... 256
Example: Build hub control ... 259
Supported examples ... 260
Supported platforms ... 260
Enable CloudWatch Logs ... 260
Prerequisites ... 261
Setup Hub SDK log configurations ... 261
Supported Zigbee and Z-Wave device types ... 263
Run managed integrations on Raspberry Pi ... 265
Sonoff Zigbee firmware flash ... 266
Managed Integrations Hub SDK image on Raspberry Pi ... 268
Managed Integrations Hub SDK Docker container on Raspberry Pi ... 274
Managed Integrations demo application ... 278
Offboard managed integrations hub ... 280
Overview of Hub SDK offboard process ... 280
Prerequisites ... 282
Hub SDK offboard process ... 283
After offboarding Hub SDK ... 286
Protocol-specific middleware ... 287
Middleware architecture ... 288
End-to-end middleware command flow example ... 289
Middleware code organization ... 291
Integrate middleware with SDK ... 297
End device SDK ... 300
What is the End device SDK? ... 300
Architecture and components ... 301
Provisionee ... 302
Provisionee workflow ... 303
Set environment variables ... 303
Register a custom endpoint ... 303
Create a provisioning profile ... 304

Create a managed thing ... 304
SDK user Wi-Fi provisioning ... 305
Fleet provisioning by claim ... 306
Managed thing capabilities ... 306
OTA updates ... 306
OTA architecture overview ... 306
Prerequisites ... 307
Implement Over-the-Air (OTA) tasks ... 308
OTA task configurations setup ... 311
Apply configuration settings to OTA tasks ... 312
Monitor OTA notifications ... 313
Process job documents ... 314
Implement OTA agent ... 314
Data model code generator ... 315
Code generation process ... 316
Environment setup ... 319
Generate code for devices ... 320
Low level C-Function APIs ... 322
OnOff cluster API ... 323
Service-device interactions ... 325
Handling remote commands ... 325
Handling unsolicited events ... 326
Get started with End device SDK ... 327
Port the End device SDK ... 339
Technical reference ... 342
Security ... 345
Data protection ... 346
Data encryption at rest for Managed Integrations ... 347
Identity and access management ... 353
Audience ... 353
Authenticating with identities ... 353
Managing access using policies ... 355
AWS managed policies ... 356
How Managed Integrations works with IAM ... 360
Identity-based policy examples ... 365
Troubleshooting ... 368

Using service-linked roles ... 370
Use AWS Secrets Manager for data protection for C2C workflows ... 373
How managed integrations uses secrets ... 374
How to create a secret ... 374
Grant access for managed integrations for AWS IoT Device Management to retrieve the secret ... 374
Compliance validation ... 376
Use Managed Integrations with interface VPC endpoints ... 376
VPC endpoint considerations ... 377
Creating VPC endpoints ... 378
Testing VPC endpoints ... 379
Access control ... 380
Pricing ... 382
Limitations ... 382
Connect to managed integrations for AWS IoT Device Management FIPS endpoints ... 382
Control plane endpoints ... 382
Logging configurations ... 384
Supported resource types ... 384
Creating event log configurations ... 384
Updating event log configurations ... 385
Configuring runtime logs ... 386
Setting up alarms on error logs ... 386
Monitoring ... 388
CloudTrail logs ... 388
Management events in CloudTrail ... 389
Event examples ... 390
Document history ... 394

What is managed integrations for AWS IoT Device Management?

Managed Integrations for AWS IoT Device Management helps IoT solution providers unify the control and management of IoT devices from hundreds of manufacturers. You can use managed integrations to automate device setup workflows and support interoperability across many devices, regardless of device vendor or connectivity protocol. With managed integrations, you can use a single user interface and set of APIs to control, manage, and operate a range of devices.

Topics

- Supported Regions
- Are you a first-time Managed Integrations user?
- Managed Integrations overview
- Managed Integrations terminology

Supported Regions

Managed Integrations for AWS IoT Device Management is supported in the following Regions:

- Canada (Central)
- Europe (Ireland)

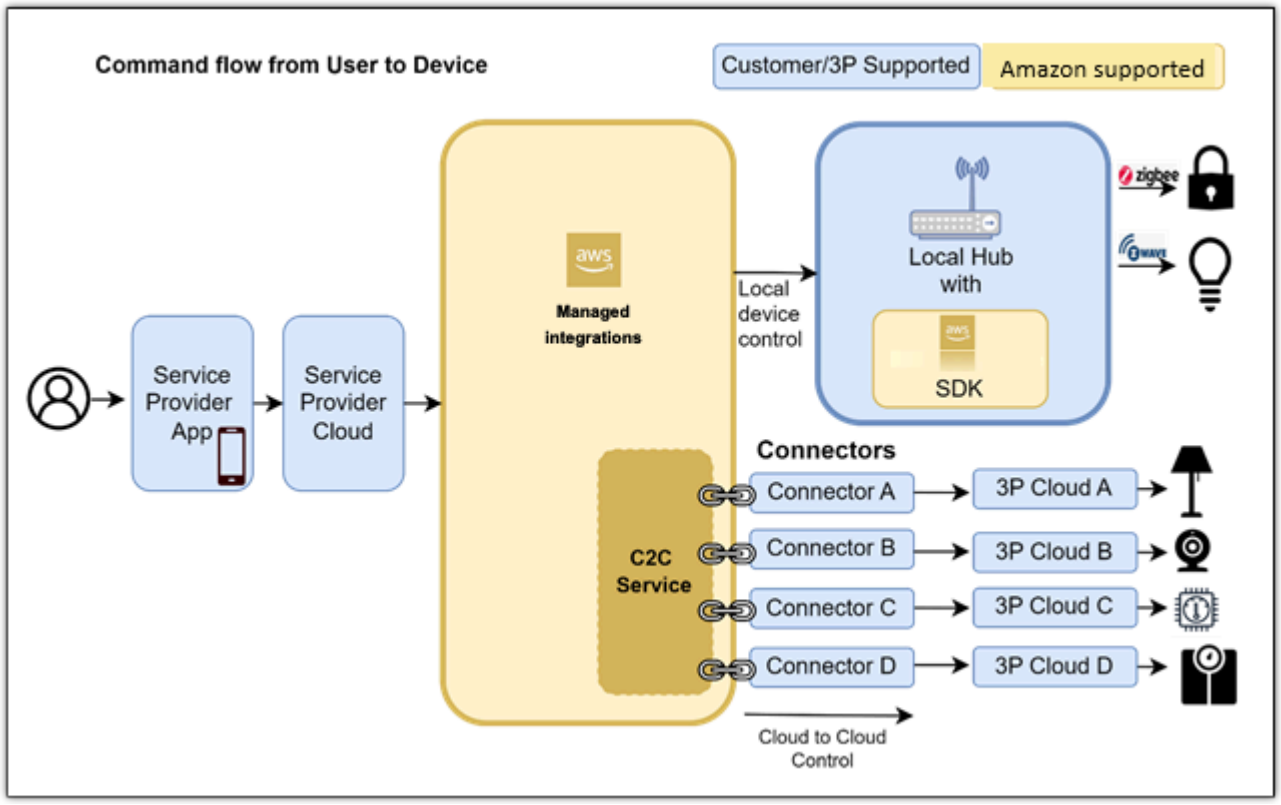
Are you a first-time Managed Integrations user?

If you are a first-time user of managed integrations, we recommend that you begin by reading the following sections:

- Set up Managed Integrations
- Get started with managed integrations for AWS IoT Device Management

Managed Integrations overview

The following image provides a high-level overview of Managed Integrations



Managed Integrations terminology

Within Managed Integrations, there are many concepts and terms critical to understand for managing your own device implementations. The following sections outline those key concepts and terms to provide a better understanding of Managed Integrations.

General Managed Integrations terminology

An important concept to understand for Managed Integrations is a *Managed Thing* compared to an AWS IoT Core thing.

- *AWS IoT Core thing*: An AWS IoT Core Thing is an AWS IoT Core construct that provides the digital representation. Developers are expected to manage policies, data storage, rules, actions, MQTT topics, and delivery of device state to the data storage. For more information on what an AWS IoT Core thing is, see Managing devices with AWS IoT.
- *Managed Integrations Managed Thing*: With a Managed Thing, we provide an abstraction to simplify device interactions and do not require the developer to create items such as rules, actions, MQTT Topics, and policies.

Cloud-to-cloud terminology

Physical devices that integrate with Managed Integrations may originate from a third-party cloud provider. To onboard those devices to Managed Integrations and communicate with the third-party cloud provider, the following terminology covers some of the key concepts supporting those workflows:

- *Cloud-to-cloud (C2C) connector*: A C2C connector establishes a connection between Managed Integrations and the third-party cloud provider.
- *Third-party cloud provider*: For devices that are manufactured and managed outside of Managed Integrations, a third-party cloud provider enables control of these devices for the end user and Managed Integrations communicates with the third-party cloud provider for various workflows such as device commands.

Data model terminology

Managed Integrations uses data models for organizing data and end-to-end communication between your devices. The following terminology covers some of the key concepts for understanding those two data models:

- **Device:** An entity representing a physical device (such as a video doorbell) which has multiple nodes working together to provide a complete feature set.
- **Endpoint:** An endpoint encapsulates a standalone feature (ringer, motion detection, lighting in a video doorbell).
- **Capability:** An entity representing components which are needed to make a feature available in an endpoint (button or a light and chime in the bell feature of video doorbell).
- **Action:** An entity representing an interaction with a capability of a device (ring the bell or view who's at the door).
- **Event:** An entity representing an event from a capability of a device. A device can send an event to report an incident/alarm, an activity from a sensor etc. (e.g. there is knock/ring on the door).
- **Property:** An entity representing a particular attribute in device state (bell is ringing, porch light is on, camera is recording).
- **Data Model:** The data layer corresponds to the data and verb elements that help support the functionality of the application. The Application operates on these data structures when there is an intent to interact with the device. For more information, see connectedhomeip on the *GitHub* website.

- **Schema:** A schema is a representation of the data model in JSON format.

Set up Managed Integrations

The following sections guide you through initial setup for using managed integrations for AWS IoT Device Management.

Topics

- Sign up for an AWS account

Sign up for an AWS account

To get started with AWS, you need an AWS account. There are two ways to sign up for AWS. If you want complete control over account configuration and where your content is stored, or if you have regulated workloads such as HIPAA or FedRAMP, use Sign up for AWS (advanced). If you are a new user and want to use a login you already own to sign in to AWS and create an AWS environment with preconfigured defaults in select AWS Regions, use Sign up for AWS.

Get started with managed integrations for AWS IoT Device Management

The following sections outline the steps that you need to take to start using Managed Integrations.

Topics

- Device types
- Configure encryption key
- Onboarding techniques

Device types

Managed Integrations manages many types of devices. Each device is within one of the following three categories:

- *Direct-connected devices*: This type of device directly connects to an Managed Integrations endpoint. Typically, these devices are built and managed by device manufacturers that include the Managed Integrations end device SDK for the direct connectivity.
- *Hub-connected devices*: These devices connect to Managed Integrations through a hub running the managed integrations Hub SDK, which manages device discovery, onboarding, and control functions. End-users can onboard these devices using button press initiation or barcode scanning.

  The following two workflows are supported for onboarding a hub-connected device:

  - An end user initiated button press to start device discovery
  - Barcode-based scanning to perform the device association
- *Cloud-to-cloud (C2C) devices*: These are devices that are designed and managed by vendors that maintain their own cloud infrastructure and branded mobile applications for device control. Managed integrations customers can access a catalog of pre-built C2C connectors or create their own, to develop IoT solutions that work with multiple third-party vendor clouds through a unified interface.

  When the end user powers on a C2C device for the first time, it must be provisioned with its respective third-party cloud provider for Managed Integrations to obtain its device capabilities

and metadata. After completing that provisioning workflow, Managed Integrations can communicate with the cloud device and the third-party cloud provider on behalf of the end user.

> **Note**
>
> A hub is not a specific device type as listed above. Its purpose is serving the role as a controller of smart home devices and facilitating a connection between Managed Integrations and third-party cloud providers. It can serve the role as both a device type as listed above and as a hub.

Configure encryption key

Security is of paramount importance for data routed between the end user, Managed Integrations, and third-party clouds. One of the methods we support to protect your device data is end-to-end encryption leveraging a secure encryption key for routing your data.

As a customer of Managed Integrations, you have the following two options for using encryption keys:

- Use the default Managed Integrations-managed encryption key.
- Provide an AWS KMS key that you created.

For more information on the AWS KMS service, see Key management service (KMS)

Calling the PutDefaultEncryptionConfiguration API in the *Managed Integrations API Reference Guide* grants you access to update which encryption key option you want to use. By default, managed integrations uses the default managed integrations managed encryption key. You can update your encryption key configuration at any time using the PutDefaultEncryptionConfiguration API.

Additionally, calling the GetDefaultEncryptionConfiguration API command returns information about the encryption configuration for the AWS account in the default or specified region.

Onboarding techniques

Listed below are the types of onboarding:

Direct-connected device onboarding

See Provisionee for steps to onboard a direct connected device.

Hub onboarding

See Onboard your hubs to Managed Integrations for steps to onboard the hub.

Hub-connected device onboarding

See Onboard devices and operate them in hub for steps to onboard a hub connected device.

Cloud-to-cloud device onboarding

See Use a C2C (Cloud-to-Cloud) connector for steps to onboard a cloud device from a third-party cloud vendor to managed integrations.

Device provisioning

Device provisioning facilitates the device onboarding process, oversees the entire device lifecycle, and establishes a centralized repository for device information that is accessible to other aspects of Managed Integrations. Managed Integrations provides a unified interface for managing various device types, accommodating first-party customer devices directly connected through a device software development kit (SDK) or commercial-off-the-shelf (COTS) devices indirectly linked via a hub device.

Each device, regardless of the device type, in Managed Integrations has a globally unique identifier called a `managedThingId`. This identifier is used in the onboarding and management of the device for the entire device lifecycle. It is fully managed by Managed Integrations and unique to that specific device across all of Managed Integrations in all AWS Regions. When a device is initially added to Managed Integrations, this identifier is created and attached to the managed thing in Managed Integrations. A managed thing is a digital representation of the physical device within Managed Integrations to mirror all device metadata of the physical device. For third-party devices, they may have their own, separate unique identifier specific to their third-party cloud in addition to the `managedThingId` stored in Managed Integrations representing the physical device.

Devices being provisioned can have different statuses depending on what stage of the onboarding flow they are in. The following list describes each provisioning status:

- ACTIVATED: The device has been found and command and control is available.
- DISCOVERED: The device has been found but command and control is not yet available.
- UNASSOCIATED: The managed thing has been created but requires further actions to be discovered. It is not reachable from the AWS Cloud or AWS IoT Managed Integrations controllers (hubs)
- PRE_ASSOCIATED: The managed thing has been created and is ready for automatic discovery once powered on or connected. It is not reachable from the AWS Cloud or AWS IoT Managed Integrations controllers (hubs).
- DELETE_IN_PROGRESS: Asynchronous deletion process started.
- DELETED: The device has been deleted from the AWS Cloud.
- ISOLATED: A previously discovered or activated managed thing that is no longer reachable. For example, a device for a third-party cloud whose connector associations have all been deleted.

The following onboarding flow is for provisioning your hub with Managed Integrations:

Onboard your hubs to Managed Integrations: Setup core provisioner and protocol-specific plugins that work together to handle device authentication, communication, and setup.

The following onboarding flows are provided for provisioning your hub connected devices with Managed Integrations:

- Simple setup (SS): The end user powers on the IoT device and scans its QR code using the device manufacturer application. The device is then enrolled onto the Managed Integrations cloud and connects to the IoT hub.
- Zero-touch setup (ZTS): The device is pre-associated upstream in the supply chain. For example, instead of end-users scanning the device QR code, this step is completed earlier to pre-link the device to the customer accounts.
- User guided setup (UGS): The end user powers on the device and follows interactive steps to onboard it to Managed Integrations. This might include pressing a button on the IoT hub, using a device manufacturer app, or pressing buttons on both the hub and device. You can use this method if Simple setup fails.

> **Note**
>
> The device provisioning workflow in Managed Integrations is agnostic of the onboarding requirements for a device. Managed Integrations provides a streamlined user interface for onboarding and managing a device, regardless of the device type or device protocol.

Device and device profile lifecycle

Managing the lifecycle of your devices and device profiles ensures your fleet of devices are secure and running efficiently.

Topics

- Device
- Device profile

Device

During initial onboarding, Managed Integrations creates a digital twin of your physical device called a *Managed Thing*. The Managed Thing has a `managedThingID` that provides a global unique identifier to identify the device in Managed Integrations across all regions. The device pairs with the local hub during provisioning for real-time communication with Managed Integrations or a third-party cloud for third-party devices. A device is also associated with an owner as identified by the `owner` parameter in the public APIs for a Managed Thing such as `GetManagedThing`. The device is linked to the corresponding device profile based on the type of device.

> **Note**
>
> A physical device may have multiple records if it is provisioned multiple times under different customers.

The device lifecycle starts with the creation of the Managed Thing in Managed Integrations using the `CreateManagedThing` API and ends when the customer deletes the Managed Thing using the `DeleteManagedThing` API. The lifecycle of a device is managed by the following public APIs:

- `CreateManagedThing`
- `ListManagedThings`
- `GetManagedThing`
- `UpdateManagedThing`
- `DeleteManagedThing`

Device profile

A device profile represents a specific type of device such as a light bulb or doorbell. It is associated with a manufacturer and contains the capabilities of the device. The device profile stores the authentication materials needed for device connectivity setup requests with Managed Integrations. The authentication materials used are the device bar code.

During the device manufacturing process, the manufacturer can register their device profiles with Managed Integrations. This enables the manufacturer to obtain the necessary materials for the devices from Managed Integrations during the onboarding and provisioning workflows. The metadata from the device profile is stored on the physical device or printed on the device labeling. The lifecycle of the device profile ends when the manufacturer deletes it in Managed Integrations.

Data models

A data model represents the organizational hierarchy of how data is organized within a system. Additionally, it supports end-to-end communication across your entire device implementation. For Managed Integrations, there are two data models used. The Managed Integrations data model and the AWS implementation of the Matter Data Model. They have similarities, but also have subtle differences that are outlined in the following topics.

For third-party devices, both data models are used for communication between the end user, Managed Integrations, and the third-party cloud provider. To translate messages such as device commands and device events from the two data models, the Cloud-to-Cloud Connector functionality is leveraged.

Topics

- Managed Integrations data model
- AWS implementation of the Matter data model
- Data model schemas

Managed Integrations data model

The Managed Integrations data model manages all communication between the end user and Managed Integrations.

Device Hierarchy

The `endpoint` and `capability` data elements are used to describe a device in the Managed Integrations data model.

`endpoint`

The `endpoint` represents the logical interfaces or services offered by the feature.

```
{
    "endpointId": { "type":"string" },
    "capabilities": Capability[]
}
```

`Capability`

The `capability` represents the device capabilities.

```
{
    "$id": "string",                // Schema identifier (e.g. /schema-versions/
capability/matter.OnOff@1.4)
    "name": "string",               // Human readable name
    "version": "string",            // e.g. 1.0
    "properties": Property[],
    "actions": Action[],
    "events": Event[]
}
```

For the `capability` data element, there are three items that comprise that item: `property`, `action`, and `event`. They can be used to interact with and monitor the device.

- **Property**: States that are held by the device, such as the current brightness level attribute of a dimmable light.
  - ```
    {
        "name":                      // Property Name is outside of Property Entity
        "value": Value,              // value represented in any type e.g. 4, "A", []
        "lastChangedAt": Timestamp   // ISO 8601 Timestamp upto milliseconds yyyy-MM-
    ddTHH:mm:ss.ssssssZ
        "mutable": boolean,
        "retrievable": boolean,
        "reportable": boolean

    }
    ```
- **Action**: Tasks that may be performed, such as locking a door on a door lock. Actions may generate responses and results.
  - ```
    {
        "name": { "$ref": "/schema-versions/definition/aws.name@1.0" }, //required
        "parameters": Map<String name, JSONNode value>,
        "responseCode": HTTPResponseCode,
        "errors": {
            "code": "string",
            "message": "string"
        }
    }
    ```

- **Event**: Essentially a record of past state transitions. While `property` represent the current states, events are a journal of the past, and include a monotonically increasing counter, a timestamp, and a priority. They enable capturing state transitions, as well as data modeling that is not readily achieved with `property`.
  - ```
    {
        "name": { "$ref": "/schema-versions/definition/aws.name@1.0" },        //
    required
        "parameters": Map<String name, JSONNode value>
    }
    ```

AWS implementation of the Matter data model

The AWS implementation of the Matter Data Model manages all communication between Managed Integrations and third-party cloud providers.

For more information, see Matter Data Model: Developer Resources.

Device Hierarchy

There are two data elements used to describe a device: `endpoint` and `cluster`.

`endpoint`

The `endpoint` represents the logical interfaces or services offered by the feature.

```
{
    "id": { "type":"string"},
    "clusters": Cluster[]
}
```

`cluster`

The `cluster` represents the device capabilities.

```
{
    "id": "hexadecimalString",
   "revision": "string"        // optional
    "attributes": AttributeMap<String attributeId, JSONNode>,
   "commands": CommandMap<String commandId, JSONNode>,
```
```

```
   "events": EventMap<String eventId, JsonNode>
}
```

For the `cluster` data element, there are three items that comprise that item: `attribute`, `command`, and `event`. They can be used to interact with and monitor the device.

- **Attribute**: States that are held by the device, such as the current brightness level attribute of a dimmable light.
  - ```
    {
        "id" (hexadecimalString): (JsonNode) value
    }
    ```
- **Command**: Tasks that may be performed, such as locking a door on a door lock. Commands may generate responses and results.
  - ```
    "id": {
        "fieldId": "fieldValue",
        ...
        "responseCode": HTTPResponseCode,
        "errors": {
            "code": "string",
            "message": "string"
        }
    }
    ```
- **Event**: Essentially a record of past state transitions. While `attributes` represent the current states, events are a journal of the past, and include a monotonically increasing counter, a timestamp, and a priority. They enable capturing state transitions, as well as data modeling that is not readily achieved with `attributes`.
  - ```
    "id": {
        "fieldId": "fieldValue",
        ...
    }
    ```

Data model schemas

Managed Integrations supports two schema types: *capability* and *type definition*. If you are creating a custom data model, you use a JSON schema document to define either type of schema. Each schema document has a limit of 50,000 characters.
    ```

Capability schema

A *capability* is a fundamental building block that represents specific functionalities within an endpoint. With capabilities, you can model device states and behaviors using properties, actions, and events. *Properties* enable you to model the device's state attributes flexibly with any declarative data type. *Actions* and *events* model the behavior of the device, including commands that it can execute and signals that it can report.

The following displays a high-level structure of a capability schema.

```
Capability
  |
  |-- Action
  |-- Event
  |-- Property
```

Action

An entity representing an interaction with a capability of a device. For example, ring the bell or view who is at the door.

Event

An entity representing an event from a capability of device. A device can send an event to report an incident, alarm, or an activity from a sensor such as a knock at the door.

Property

An entity representing a particular attribute in the state of the device. For example, a bell is ringing or the porch light is on

Each capability includes a unique namespaced identifier, version information, and a description of its purpose. The schema document uses semantic versioning to maintain backward compatibility while enabling new features.

For more information, see Schema for capability definitions.

Type definition schema

A type definition is a declarative structured data type that enables reusability and composability. It defines how information should be formatted and constrained. Use type definitions to create standardized data formats across your IoT solution.

Each type definition includes:

- A unique namespaced identifier
- Title
- Description
- Properties that define data formatting and constraints

Types can be either simple primitives, such as integers or strings with defined limits, or complex structures such as enumerations or custom objects with multiple fields. Type definitions use JSON schema syntax to specify constraints including minimum and maximum values, string lengths, and allowable patterns.

For more information, see Schema for type definitions.

Schema for capability definitions

A capability is documented using a declarative JSON document that provides a clear contract for how the capability should function within the system.

For a capability, the mandatory elements are `$id`, `name`, `extrinsicId`, `extrinsicVersion` and at least one element in at least one of the following sections:

- `properties`
- `actions`
- `events`

The optional elements in a capability are `$ref`, `title`, `description`, `version`, `$defs` and `extrinsicProperties`. For a capability, `$ref` must refer to `aws.capability`.

The following sections detail the schema used for capability definitions.

$id (mandatory)

The $id element identifies the schema definition. It must follow this structure:

- Start with the `/schema-versions/` URI prefix
- Include the `capability` schema type
- Use a forward slash (/) as a URI path separator

- Include the schema identity, with fragments separated by periods (`.`)
- Use the `@` character to separate the schema ID and version
- End with the semver version, using periods (`.`) to separate version fragments

The schema identity must start with a root namespace that is 3-12 characters long, followed by an optional sub-namespace and name.

The semver version includes a MAJOR version (up to 3 digits), a MINOR version (up to 3 digits), and an optional PATCH version (up to 4 digits).

> **Note**
>
> You can't use the reserved namespaces `aws` or `matter`

Example $id

```
/schema-version/capability/aws.Recording@1.0
```

$ref

The `$ref` element references an existing capability within the system. It follows the same constraints as the `$id` element.

> **Note**
>
> A type definition or capability must exist with the value provided in the `$ref` file.

Example $ref

```
/schema-version/definition/aws.capability@1.0
```

name (mandatory)

The name element is a string representing the entity name in the schema document. It often contains abbreviations and must follow these rules:

- Contain only alphanumeric characters, periods (.), forward slashes (/), hyphens (-), and spaces
- Start with a letter
- Maximum of 64 characters

The name element is used in the Amazon Web Services console UI and documentation.

Example names

```
Door Lock
On/Off
Wi-Fi Network Management
PM2.5 Concentration Measurement
RTCSessionController
Energy EVSE
```

title

The title element is a descriptive string for the entity represented by the schema document. It can contain any characters and is used in documentation. The maximum length for a capability title is 256 characters.

Example titles

```
Real-time Communication (RTC) Session Controller
Energy EVSE Capability
```

description

The `description` element provides a detailed explanation of the entity represented by the schema document. It can contain any characters and is used in documentation. The maximum length for a capability description is 2048 characters

Example description

```
Electric Vehicle Supply Equipment (EVSE) is equipment used to charge an Electric
 Vehicle (EV) or Plug-In Hybrid Electric Vehicle.
           This capability provides an interface to the functionality of Electric
 Vehicle Supply Equipment (EVSE) management.
```

version

The `version` element is optional. It is a string that represents the version of the schema document. It has the following constraints:

- Uses semver format, with following version fragments separated by `.` (periods).
  - `MAJOR` version, maximum of 3 digits
  - `MINOR` version, maximum of 3 digits
  - `PATCH` version (optional), maximum of 4 digits
- The length can be between 3 and 12 characters.

Example versions

```
1.0
```

```
1.12
```

```
1.4.1
```

Working with capability versions

A capability is an immutable versioned entity. Any change is expected to create a new version. The system uses semantic versioning with MAJOR.MINOR.PATCH format, where:

- MAJOR version increases when making backward incompatible API changes
- MINOR version increases when adding functionality in a backward-compatible manner
- PATCH version increases when making minor non-impacting additions in the capability.

The capabilities derived from Matter clusters are baselined from version 1.4 and each Matter release is expected to be imported into the system. As the Matter version consumes both MAJOR and MINOR levels of semver, managed integrations can only use PATCH versions.

When you add PATCH versions for Matter, be sure to take into account that that Matter uses sequential revisions. All PATCH versions must comply with the revision documented in the Matter specification and these must be backward compatible.

To get any backward-incompatibly issues fixed, you must work with Connectivity Standards Alliance (CSA) to solve those in the specification and get a new revision released.

AWS-managed capabilities were released with an initial version of `1.0`. With these, all three levels of version can be used.

extrinsicVersion (mandatory)

This is a string representing a version managed outside of the AWS IoT system. For Matter capabilities, `extrinsicVersion` maps to `revision`

It is represented as a stringified integer value, and the length can be from 1 to 10 numerical digits.

Example versions

```
7
```

```
1567
```

extrinsicId (mandatory)

The `extrinsicId` element represents an identifier managed outside of the Amazon Web Services IoT system. For Matter capabilities, it maps to `clusterId`, `attributeId`, `commandId`, `eventId`, or `fieldId`, depending on the context.

The `extrinsicId` can be either a stringified decimal integer (1-10 digits) or a stringified hexadecimal integer (0x or 0X prefix, followed by 1-8 hexadecimal digits).

> **Note**
>
> For AWS, the Vendor ID (VID) is 0x1577, and for Matter, it is 0. The system ensures that custom schemas don't use these reserved VIDs for capabilities.

Example extrinsicIds

```
0018
0x001A
```

```
0x15771002
```

$defs

The `$defs` section is a map of sub-schemas that can be referenced within the schema document as allowed by the JSON schema. In this map, the key is used in the local reference definitions and the value provides the JSON schema.

> **Note**
>
> The system only enforces that `$defs` is a valid map and that each sub-schema is a valid JSON schema. No additional rules are enforced.

Follow these constraints when working with definitions:

- Use only URI-friendly characters in definition names
- Ensure each value is a valid sub-schema
- Include any number of sub-schemas that fit within the schema document size limits

extrinsicProperties

The `extrinsicProperties` element contains a set of properties defined in an external system but maintained within the data model. For Matter capabilities, it maps to different unmodeled or partially modeled elements within ZCL cluster, attribute, command, or event.

Extrinsic Properties must follow these constraints:

- Property names must be alphanumeric without spaces or special characters
- Property values can be any JSON schema values
- Maximum of 20 properties

The system supports various `extrinsicProperties`, including `access`, `apiMaturity`, `cli`, `cliFunctionName`, and others. These properties facilitate ACL to AWS (and vice versa) data model transformations.

> **Note**
>
> Extrinsic properties are supported for the `action`, `event`, `property`, and `struct` fields elements of a capability, but not for the capability or cluster itself.

System-supported extrinsic properties

The system tracks the following unmodeled or partially modeled cluster, attribute, command, or event attributes as `extrinsicProperties` during transformations to or from ZCL:

`access`

Each access object contains the following:

- op - Operation modeled as an `enum` with values: `read`, `write`, or `invoke`
- `privilege` - Privilege modeled as an `enum` with values: `view`, `proxy_view`, `operate`, `manage`, or `administer`
- `role` - Unbounded string representing an operator role

`apiMaturity`

An unbounded plain string representing the maturity level. This is modeled in ZCL as an `enum` with values: `stable`, `provisional`, `internal`, or `deprecated`

`side`

Modeled as an enum with values: `either`, `server`, and `client`

Boolean properties

The following properties are boolean flags:

- `isFabricScoped`
- `isFabricSensitive`
- `mustUseAtomicWrite`
- `mustUseTimedInvoke`

String properties

The following properties are represented as unbounded strings:

- `cli`

- `cliFunctionName`
- `functionName`
- `group`
- `introducedIn`
- `manufacturerCode`
- `noDefaultImplementation`
- `presentIf`
- `priority`
- `removedIn`
- `reportableChange`
- `reportMinInterval`
- `reportMaxInterval`
- `restriction`
- `storage`

Transformation considerations

For ZCL transformations, `extrinsicProperties` are stored in a map without processing. Custom schemas that use discovery don't undergo ZCL transformation. However, if you plan to implement ZCL transformations for custom schemas in the future, you must model all unbounded plain string type `extrinsicProperties` and define constraints such as enums, patterns (regex), and length. This preparation ensures proper handling of these properties during transformation.

In contrast, for AWS to connector transformations, `extrinsicProperties` are not included at all, as these details are not required in the connector format.

Properties

Properties represent device-managed states of the capability. Each state is defined as a key-value pair, where the key describes the name of the state and the value describes the definition of the state.

When working with properties, follow these constraints:

- Use only alphanumeric characters in property names, without spaces or special characters

- Include any number of properties that fit within the schema document size limits

Working with properties

A property within a capability is a fundamental element that represents a specific state of a device powered by managed integrations. It represents the device's current condition or configuration. By standardizing how these properties are defined and structured, smart home systems ensure that devices from different manufacturers can communicate effectively, creating a seamless and interoperable experience.

For a capability property, the mandatory elements are `extrinsicId` and `value`. The optional elements in a capability property are `description`, `retrievable`, `mutable`, `reportable` and `extrinsicProperties`.

Value

An unbounded structure that allows builders to put any JSON-schema compliant constraints to define the data type of this property.

When defining values, follow these constraints:

- For simple types, use `type` and any other native JSON-schema constraints such as `maxLength` or `maximum`
- For composite types, use `oneOf`, `allOf`, or `anyOf`. The system doesn't support the `not` keyword
- To refer to any global type, use `$ref` with a valid discoverable reference
- For nullability, follow OpenAPI type schema definition by providing the nullable attribute with a boolean flag (`true` if null is an allowed value)

Example:

```
{
    "$ref": "/schema-versions/definition/matter.uint16@1.4",
    "nullable": true,
    "maximum": 4096
}
```

Retrievable

A boolean describing if the state is readable or not.

The readability aspect of the state is deferred to the device's implementation of the capability. The device decides if a given state is readable or not. This aspect of the state is not yet supported to be reported in the capability report and hence not enforced within the system.

Example: `true` or `false`

Mutable

A boolean describing if the state is writable or not.

The writability aspect of the state is deferred to the device's implementation of the capability. The device decides if a given state is writable or not. This aspect of the state is not yet supported to be reported in the capability report and hence not enforced within the system.

Example: `true` or `false`

Reportable

A boolean describing if the state is reported by the device when there is a change in the state.

The reportability aspect of the state is deferred to the device's implementation of the capability. The device decides if a given state is reportable or not. This aspect of the state is not yet supported to be reported in the capability report and hence not enforced within the system.

Example: `true` or `false`

Actions

Actions are schema-managed operations that follow a request-response model. Each action represents a device-implemented operation.

Follow these constraints when implementing actions:

- Include only unique actions in the actions array
- Include any number of actions that fit within the schema document size limits

Working with actions

An action is a standardized way to interact with and control device capabilities in a managed integration system. It represents a specific command or operation that can be executed on a device, complete with a structured format to model any necessary request or response parameters. These

actions serve as the bridge between user intentions and device operations, enabling consistent and reliable control across different types of smart devices.

For an action, the mandatory elements are `name` and `extrinsicId`. The optional elements are `description`, `extrinsicProperties`, `request` and `response`.

Description

The description has a maximum length constraint of 1536 characters.

Request

The request section is optional and can be omitted if there are no request parameters. If omitted, the system supports sending a request without any payload by just using the name of `Action`. This is used in simple actions, like turning a light on or off.

The complex actions need additional parameters. For instance, a request to stream camera footage might include parameters about the streaming protocol to use or whether to send the stream to a specific display device.

For an action request, the mandatory element is `parameters`. The optional elements are `description`, `extrinsicId`, and `extrinsicProperties`.

Request description

The description follows the same format as section 3.5, with a maximum length of 2048 characters.

Response

In managed integrations, for any action request sent through the SendManagedThingCommand API, the request reaches the device and expects an asynchronous response back. The action response defines the structure of this response.

For an action request, the mandatory element is `parameters`. The optional elements are `name`, `description`, `extrinsicId`, `extrinsicProperties`, `errors` and `responseCode`.

Response description

The description follows the same format as description, and has a maximum length of 2048 characters.

Response name

The name follows the same format as name (mandatory), with these additional details:

- The conventional name of a response is derived by appending `Response` to the action name.
- If you want to use a different name, you can provide it in this `name` element. If a `name` is provided in the response, then this value takes higher precedence than the conventional name.

Errors

An unbounded array of unique messages provided in the response, if there are errors while processing the request.

Constraints:

- A message item is declared as a JSON object with the following fields:
  - `code`: A string containing alphanumeric characters and _ (underscores), with a length between 1 to 64 characters
  - `message`: An unbounded string value

Example Error message example

```
"errors": [
   {
      "code": "AD_001",
      "message": "Unable to receive signal from the sensor. Please check connection
 with the sensor."
   }
]
```

Response code

An integer code displaying how the request was handled. We recommend that the device code returns a code using the HTTP server response status code specification to allow uniformity within the system.

Constraint: An integer value ranging from 100 to 599.

Request or response parameters

The parameters section is defined as a map of name and sub-schema pairs. Any number of parameters can be defined within the request parameters, if they can fit in the schema document.

Parameter names can only contain alphanumeric characters. Spaces or any other characters are not allowed.

parameter field

The mandatory elements in a `parameter` are `extrinsicId` and `value`. The optional elements are `description` and `extrinsicProperties`.

The description element follows the same format as description, with a maximum length of 1024 characters.

`extrinsicId` and `extrinsicProperties` overrides

the `extrinsicId` and `extrinsicProperties` follow the same format as extrinsicId (mandatory) and extrinsicProperties, with these additional details:

- If an `extrinsicId` is provided in the request or response, this value takes higher precedence than the value provided at the action level. The system must use request/response level `extrinsicId` first, if missing use the action level `extrinsicId`
- If `extrinsicProperties` are provided in the request or response, these properties take higher precedence than the va value provided at the action level. The system must take the action level `extrinsicProperties` and replace the key-value pairs provided at the request/response level `extrinsicProperties`

Example extrinsicId and extrinsicProperties override example

```
{
   "name": "ToggleWithEffect",
   "extrinsicId": "0x0001",

   "extrinsicProperties": {
      "apiMaturity": "provisional",
      "introducedIn": "1.2"
   },
   "request": {
      "extrinsicProperties": {
```

```
         "apiMaturity": "stable",
         "manufacturerCode": "XYZ"
      },
      "parameters": {
         ...
      }
   },
   "response": {
      "extrinsicProperties": {
         "noDefaultImplementation": true
      },
      "parameters": {
 {
         ...
      }
   }
}
```

In the above example, the effective values for the action request would be:

```
# effective request
"name": "ToggleWithEffect",
"extrinsicId": "0x0001",
"extrinsicProperties": {
   "apiMaturity": "stable",
   "introducedIn": "1.2"
   "manufacturerCode": "XYZ"
},
"parameters": {
   ...
}

# effective response
"name": "ToggleWithEffectResponse",
"extrinsicId": "0x0001",
"extrinsicProperties": {
   "apiMaturity": "provisional",
   "introducedIn": "1.2"
   "noDefaultImplementation": true
},
"parameters": {
   ...
}
```

Built-in actions

For all capabilities, you can perform custom actions using the keywords `ReadState` and `UpdateState`. These two action keywords will act on the capability's properties defined in the data model.

ReadState

Sends a command to the `managedThing` to read the values of its state properties. Use `ReadState` as a way to force the device state to be updated.

UpdateState

Sends a command to update some of the properties.

Forcing a device state synchronization may be useful in the following scenarios:

1. The device was offline for a period of time and was not emitting events.
2. The device was just provisioned and does not yet have any state maintained in the cloud.
3. The device state is out of sync with the real state of the device.

ReadState examples

Check if the light is on or off using the SendManagedThingCommand API:

```
{
  "Endpoints": [
    {
      "endpointId": "1",
      "capabilities": [
        {
          "id": "aws.OnOff",
          "name": "On/Off",
          "version": "1",
          "actions": [
            {
              "name": "ReadState",
              "parameters": {
                "propertiesToRead": [ "OnOff" ]
              }
            }
```

```
          ]
        }
      ]
    }
  ]
}
```

Read all state properties for the `matter.OnOff` capability:

```
{
  "Endpoints": [
    {
      "endpointId": "1",
      "capabilities": [
        {
          "id": "aws.OnOff",
          "name": "On/Off",
          "version": "1",
          "actions": [
            {
              "name": "ReadState",
              "parameters": {
                "propertiesToRead": [ "*" ]
                // Use the wildcard operator to read ALL state properties for a
 capability
              }
            }
          ]
        }
      ]
    }
  ]
}
```

UpdateState example

Change the `OnTime` for a light using the SendManagedThingCommand API:

```
{
  "Endpoints": [
    {
      "endpointId": "1",
      "capabilities": [
```

```
        {
          "id": "matter.OnOff",
          "name": "On/Off",
          "version": "1",
          "actions": [
            {
              "name": "UpdateState",
              "parameters": {
                "OnTime": 5
              }
            }
          ]
        }
      ]
    }
  ]
}
```

Events

Events are schema-managed, unidirectional signals implemented by the device.

Implement events according to these constraints:

- Include only unique events in the events array
- Include any number of events that fit within the schema document size limits

Events in managed integration systems

Working with events

An event is a standardized way to proactively learn about changes to a device or its surroundings. It represents a modeled event that the device will send to the cloud to provide information about something that was modified on the device or sensed in its environment. Because these events are modeled, customers can use them in a control flow to react to specific events and the details provided inside of them.

For an event, the mandatory elements are `name` and `extrinsicId`. The optional elements are `description`, `extrinsicProperties`, and `request`.

Description

The description follows the same format as described in description, with a maximum length of 512 characters.

Request

The `request` section is optional and can be omitted if there are no request parameters. If omitted, the system supports a device sending an event request without any payload by just using the name of the Event. This is used in simple events, such as a sensor failure on a pump or if the alarm is muted on a smoke or carbon monoxide alarm.

The complex actions need additional parameters. For instance, a request to stream camera footage might include parameters about the streaming protocol to use or whether to send the stream to a specific display device.

For an event request, the mandatory element is `parameters`. There are no optional elements.

Response

Event responses are not currently supported.

Schema for type definitions

The following sections detail the schema used for type definitions.

$id

The $id element identifies the schema definition. It must follow this structure:

- Start with the `/schema-versions/` URI prefix
- Include the `definition` schema type
- Use a forward slash (`/`) as a URI path separator
- Include the schema identity, with fragments separated by periods (`.`)
- Use the `@` character to separate the schema ID and version
- End with the semver version, using periods (`.`) to separate version fragments

The schema identity must start with a root namespace that is 3-12 characters long, followed by an optional sub-namespace and name.

The semver version includes a MAJOR version (up to 3 digits), a MINOR version (up to 3 digits), and an optional PATCH version (up to 4 digits).

> **Note**
>
> You can't use the reserved namespaces `aws` or `matter`

Example $id

```
/schema-version/capability/aws.Recording@1.0
```

$ref

The $ref element references an existing type definition within the system. It follows the same constraints as the `$id` element.

> **Note**
>
> A type definition or capability must exist with the value provided in the `$ref` file.

Example $ref

```
/schema-version/definition/aws.capability@1.0
```

name

The name element is a string representing the entity name in the schema document. It often contains abbreviations and must follow these rules:

- Contain only alphanumeric characters, periods (.), forward slashes (/), hyphens (-), and spaces
- Start with a letter
- Maximum of 192 characters

The name element is used in the Amazon Web Services console UI and documentation.

Example names

```
Door Lock
On/Off
Wi-Fi Network Management
PM2.5 Concentration Measurement
RTCSessionController
Energy EVSE
```

title

The title element is a descriptive string for the entity represented by the schema document. It can contain any characters and is used in documentation.

Example titles

```
Real-time Communication (RTC) Session Controller
Energy EVSE Capability
```

description

The `description` element provides a detailed explanation of the entity represented by the schema document. It can contain any characters and is used in documentation.

Example description

```
Electric Vehicle Supply Equipment (EVSE) is equipment used to charge an Electric
 Vehicle (EV) or Plug-In Hybrid Electric Vehicle.
           This capability provides an interface to the functionality of Electric
 Vehicle Supply Equipment (EVSE) management.
```

extrinsicId

The `extrinsicId` element represents an identifier managed outside of the Amazon Web Services IoT system. For Matter capabilities, it maps to `clusterId`, `attributeId`, `commandId`, `eventId`, or `fieldId`, depending on the context.

The `extrinsicId` can be either a stringified decimal integer (1-10 digits) or a stringified hexadecimal integer (0x or 0X prefix, followed by 1-8 hexadecimal digits).

> **Note**
>
> For AWS, the Vendor ID (VID) is 0x1577, and for Matter, it is 0. The system ensures that custom schemas don't use these reserved VIDs for capabilities.

Example extrinsicIds

```
0018
0x001A
0x15771002
```

extrinsicProperties

The `extrinsicProperties` element contains a set of properties defined in an external system but maintained within the data model. For Matter capabilities, it maps to different unmodeled or partially modeled elements within ZCL cluster, attribute, command, or event.

Extrinsic Properties must follow these constraints:

- Property names must be alphanumeric without spaces or special characters
- Property values can be any JSON schema values
- Maximum of 20 properties

The system supports various `extrinsicProperties`, including `access`, `apiMaturity`, `cli`, `cliFunctionName`, and others. These properties facilitate ACL to AWS (and vice versa) data model transformations.

> **Note**
>
> Extrinsic properties are supported for the `action`, `event`, `property`, and `struct` fields elements of a capability, but not for the capability or cluster itself.

System-supported extrinsic properties

The system tracks the following unmodeled or partially modeled cluster, attribute, command, or event attributes as `extrinsicProperties` during transformations to or from ZCL:

`access`

Each access object contains the following:

- op - Operation modeled as an `enum` with values: `read`, `write`, or `invoke`
- `privilege` - Privilege modeled as an `enum` with values: `view`, `proxy_view`, `operate`, `manage`, or `administer`
- `role` - Unbounded string representing an operator role

`apiMaturity`

An unbounded plain string representing the maturity level. This is modeled in ZCL as an `enum` with values: `stable`, `provisional`, `internal`, or `deprecated`

`side`

Modeled as an enum with values: `either`, `server`, and `client`

Boolean properties

The following properties are boolean flags:

- `isFabricScoped`
- `isFabricSensitive`
- `mustUseAtomicWrite`
- `mustUseTimedInvoke`

String properties

The following properties are represented as unbounded strings:

- `cli`
- `cliFunctionName`
- `functionName`
- `group`
- `introducedIn`
- `manufacturerCode`
- `noDefaultImplementation`
- `presentIf`
- `priority`

- `removedIn`
- `reportableChange`
- `reportMinInterval`
- `reportMaxInterval`
- `restriction`
- `storage`

Transformation considerations

For ZCL transformations, `extrinsicProperties` are stored in a map without processing. Custom schemas that use discovery don't undergo ZCL transformation. However, if you plan to implement ZCL transformations for custom schemas in the future, you must model all unbounded plain string type `extrinsicProperties` and define constraints such as enums, patterns (regex), and length. This preparation ensures proper handling of these properties during transformation.

In contrast, for AWS to connector transformations, `extrinsicProperties` are not included at all, as these details are not required in the connector format.

Building and using type definitions in capability schema documents

All elements in the schemas resolve to type definitions. These type definitions are either *primitive type definitions* (such as booleans, strings, numbers) or *namespaced type definitions* (type definitions built from primitive type definitions for convenience).

When you define a custom schema, you can use both primitive definitions and namespace type definitions.

Contents

- Primitive type definitions
  - Booleans
  - Integer type support
  - Numbers
  - Strings
  - Nulls
  - Arrays

- Objects
- Namespaced type definitions
  - matter types
  - aws types
    - Bitmap type definition
    - Enum type definition

Primitive type definitions

Primitive type definitions are the building blocks for all type definitions defined in managed integrations. All namespace definitions, including custom type definitions, resolve to a primitive type definition either through the `$ref` keyword or the `type` keyword.

All primitive types are nullable by using the `nullable` keyword, and you can identify all primitive types by using the `type` keyword.

Booleans

Boolean types support default values.

Sample definition:

```
{
    "type" : "boolean",
    "default" : "false",
    "nullable" : true
}
```

Integer type support

Integer types support the following:

- `default` values
- `maximum` values
- `minimum` values
- `exclusiveMaximum` values

- `exclusiveMinimum` values
- `multipleOf` values

If x is the value being validated, the following must be true:

- x ≥ `minimum`
- x > `exclusiveMinimum`
- x < `exclusiveMaximum`

> **Note**
>
> Numbers with a zero fractional part are considered integers, but floating point numbers are rejected.
>
> ```
> 1.0 // Schema-Compliant
> 3.1415926 // NOT Schema-Compliant
> ```

While you can specify both `minimum` and `exclusiveMinimum` or both `maximum` and `exclusiveMaximum`, we don't recommend using both simultaneously.

Sample definitions:

```
{
    "type" : "integer",
    "default" : 2,
    "nullable" : true,
    "maximum" : 10,
    "minimum" : 0,
    "multipleOf": 2
}
```

Alternative definition:

```
{
    "type" : "integer",
    "default" : 2,
    "nullable" : true,
```

```
    "exclusiveMaximum" : 11,
    "exclusiveMinimum" : -1,
    "multipleOf": 2
}
```

Numbers

Use the number type for any numeric type, including integers and floating point numbers.

Number types support the following:

- `default` values
- `maximum` values
- `minimum` values
- `exclusiveMaximum` values
- `exclusiveMinimum` values
- `multipleOf` values. The multiple can be a floating point number.

If `x` is the value being validated, the following must be true:

- `x` ≥ `minimum`
- `x` > `exclusiveMinimum`
- `x` < `exclusiveMaximum`

While you can specify both `minimum` and `exclusiveMinimum` or both `maximum` and `exclusiveMaximum`, we don't recommend using both simultaneously.

Sample definitions:

```
{
    "type" : "number",
    "default" : 0.4,
    "nullable" : true,
    "maximum" : 10.2,
    "minimum" : 0.2,
    "multipleOf": 0.2
}
```

Alternative definition:

```
{
    "type" : "number",
    "default" : 0.4,
    "nullable" : true,
    "exclusiveMaximum" : 10.2,
    "exclusiveMinimum" : 0.2,
    "multipleOf": 0.2
}
```

Strings

String types support the following:

- `default` values
- length constraints (must be non-negative numbers) including `maxLength` and `minLength` values
- `pattern` values for regular expressions

When you define regular expressions, the string is valid if the expression matches anywhere within the string. For example, the regular expression p matches any string containing a p, such as "apple", not just the string "p". For clarity, we recommend surrounding regular expressions with `^...$` (for example, ^p$), unless you have a specific reason not to do so.

Sample definition:

```
{
    "type" : "string",
    "default" : "defaultString",
    "nullable" : true,
    "maxLength": 10,
    "minLength": 1,
    "pattern" : "^([0-9a-fA-F]{2})+$"
}
```

Nulls

Null types accept only a single value: `null`.

Sample definition:

```
{ "type": "null" }
```

Arrays

Array types support the following:

- `default` — a list that will be used as the default value.
- `items` — JSON type definition imposed on all of the array elements.
- Length constraints (must be a non-negative number)
  - `minItems`
  - `maxItems`
- `pattern` values for Regex
- `uniqueItems` — a boolean indicating if the elements in the array need to be unique
- `prefixItems` — an array where each item is a schema that corresponds to each index of the document's array. That is, an array where the first element validates the first element of the input array, the second element validates the second element of the input array, and so on.

Sample definition:

```
{
   "type": "array",
   "default": ["1", "2"],
   "items" : {
        "type": "string",
        "pattern": "^([a-zA-Z0-9_ -/]+)$"
    },
    "minItems" : 1,
    "maxItems": 4,
    "uniqueItems" : true,
}
```

Examples of array validation:

```
//Examples:
["1", "2", "3", "4"] // Schema-Compliant
[]                   // NOT Schema-Compliant: minItems=1
```

```
["1", "1"]          // NOT Schema-Compliant: uniqueItems=true
["{"]               // NOT Schema-Compliant: Does not match the RegEx pattern.
```

Alternative definition using tuple validation:

```
{
    "type": "array",
    "prefixItems": [
        { "type": "number" },
        { "type": "string" },
        { "enum": ["Street", "Avenue", "Boulevard"] },
        { "enum": ["NW", "NE", "SW", "SE"] }
    ]
}

//Examples:
[1600, "Pennsylvania", "Avenue", "NW"]              // Schema-Compliant

// And, by default, it's also okay to add additional items to end:
[1600, "Pennsylvania", "Avenue", "NW", "Washington"] // Schema-Compliant
```

Objects

Object types support the following:

- Property constraints
  - `properties` — Define the properties (key-value pairs) of an object by using the `properties`keyword. The value of `properties` is an object, where each key is the name of a property and each value is a schema used to validate that property. Any property that doesn't match any of the property names in the `properties` keyword is ignored by this keyword.
  - `required` — By default, the properties defined by the `properties` keyword are not required. However, you can provide a list of required properties using the `required` keyword. The `required` keyword takes an array of zero or more strings. Each of these strings must be unique.
  - `propertyNames` — This keyword allows control over the RegEx pattern for property names. For example, you might want to enforce that all properties of an object have names following a specific convention.
  - `patternProperties` — This maps regular expressions to schemas. If a property name matches the given regular expression, the property value must validate against the

corresponding schema. For example, use `patternProperties` to specify that a value should match a particular schema, given a particular kind of property name.

- `additionalProperties` — This keyword controls how extra properties are handled. Extra properties are properties whose names are not listed in the properties keyword or that match any of the regular expressions in `patternProperties`. By default, additional properties are allowed. Setting this field to `false` means that no additional properties are allowed.
- `unevaluatedProperties` — This keyword is similar to `additionalProperties` except that it can recognize properties declared in subschemas. `unevaluatedProperties` works by collecting any properties that are successfully validated when processing the schemas and using those as the allowed list of properties. This allows you to do more complex things such as conditionally adding properties. Refer to the example below for more details.

- `anyOf` — This keyword's value MUST be a non-empty array. Each item of the array MUST be a valid JSON Schema. An instance validates successfully against this keyword if it validates successfully against *at least* one schema defined by this keyword's value.
- `oneOf` — This keyword's value MUST be a non-empty array. Each item of the array MUST be a valid JSON Schema. An instance validates successfully against this keyword if it validates successfully against *exactly* one schema defined by this keyword's value.

Example of required:

```
{
  "type": "object",
  "required": ["test"]
}

// Schema Compliant
{
  "test": 4
}

// NOT Schema Compliant
{}
```

PropertyNames example:

```
{
  "type": "object",
  "propertyNames": {
```

```
    "pattern": "^[A-Za-z_][A-Za-z0-9_]*$"
  }
}

// Schema Compliant
{
  "_a_valid_property_name_001": "value"
}

// NOT Schema Compliant
{
  "001 invalid": "value"
}
```

PatternProperties example:

```
{
  "type": "object",
  "patternProperties": {
    "^S_": { "type": "string" },
    "^I_": { "type": "integer" }
  }
}

// Schema Compliant
{ "S_25": "This is a string" }
{ "I_0": 42 }

// NOT Schema Compliant
{ "S_0": 42 } // Value must be a string
{ "I_42": "This is a string" } // Value must be an integer
```

AdditionalProperties example:

```
{
  "type": "object",
  "properties": {
    "test": {
        "type": "string"
    }
  },
  "additionalProperties": false
}
```

```
// Schema Compliant
{
  "test": "value"
}
OR
{}

// NOT Schema Compliant
{
    "notAllowed": false
}
```

UnevaluatedProperties example:

```
{
  "type": "object",
  "properties": {
    "standard_field": { "type": "string" }
  },
  "patternProperties": {
    "^@": { "type": "integer" } // Allows properties starting with '@'
  },
  "unevaluatedProperties": false // No other properties allowed
}

// Schema Compliant
{
  "standard_field": "some value",
  "@id": 123,
  "@timestamp": 1678886400
}
// This passes because "standard_field" is evaluated by properties,
// "@id" and "@timestamp" are evaluated by patternProperties,
// and no other properties remain unevaluated.

// NOT Schema Compliant
{
  "standard_field": "some value",
  "another_field": "unallowed"
}
// This fails because "another_field" is unevaluated and doesn't match
// the @ pattern, leading to a violation of unevaluatedProperties: false
```

AnyOf example:

```
{
  "anyOf": [
    { "type": "string", "maxLength": 5 },
    { "type": "number", "minimum": 0 }
  ]
}

// Schema Compliant
"short"
12

// NOT Schema Compliant
"too long"
-5
```

OneOf example:

```
{
  "oneOf": [
    { "type": "number", "multipleOf": 5 },
    { "type": "number", "multipleOf": 3 }
  ]
}

// Schema Compliant
10
9

// NOT Schema compliant
2  // Not a multiple of either 5 or 3
15 // Multiple of both 5 and 3 is rejected.
```

Namespaced type definitions

Namespaced type definitions are types built from primitive types. These types must follow the format *`namespace`*`.`*`typename`*`.` Managed Integrations provides predefined types under the `aws` and `matter` namespaces. You can use any namespace for custom types except the reserved `aws` and `matter` namespaces.

To find available namespaced type definitions, use the ListSchemaVersions API with the `Type` filter set to `definition`.

`matter` types

Find data types under the `matter` namespace using the ListSchemaVersions API with the `Namespace` filter set to `matter` and the `Type` filter set to `definition`.

`aws` types

Find data types under the `aws` namespace using the ListSchemaVersions API with the `Namespace` filter set to `aws` and the `Type` filter set to `definition`.

Bitmap type definition

Bitmaps have two required properties:

- `type` must be an object
- `properties` must be an object containing each bit definition. Each bit is an object with properties `extrinsicId` and `value`. Each bit's value must be an integer with a minimum value of 0 and a maximum value of at least 1.

Sample bitmap definition:

```
{
    "title" : "Sample Bitmap Type",
    "description" : "Type definition for SampleBitmap.",
    "$ref" : "/schema-versions/definition/aws.bitmap@1.0 ",
    "type" : "object",
    "additionalProperties" : false,
    "properties" : {
        "Bit1" : {
            "extrinsicId" : "0x0000",
            "value" : {
                "type" : "integer",
                "maximum" : 1,
                "minimum" : 0
            }
        },
        "Bit2" : {
            "extrinsicId" : "0x0001",
            "value" : {
```

```
                "type" : "integer",
                "maximum" : 1,
                "minimum" : 0
            }
        }
    }
}

// Schema Compliant
{
    "Bit1": 1,
    "Bit1": 0
}

// NOT Schema Compliant
{
    "Bit1": -1,
    "Bit1": 0
}
```

Enum type definition

Enums require three properties:

- `type` must be an object
- `enum` must be an array of unique strings, with a minimum of one item
- `extrinsicIdMap` is an object with properties that are the enum values. The value of each of the properties should be the extrinsic identifier that correspond to the enum value.

Sample enum definition:

```
{
    "title" : "SampleEnum Type",
    "description" : "Type definition for SampleEnum.",
    "$ref" : "/schema-versions/definition/aws.enum@1.0",
    "type" : "string",
    "enum" : [
        "EnumValue0",
        "EnumValue1",
        "EnumValue2"
    ],
    "extrinsicIdMap" : {
```

```
        "EnumValue0" : "0",
        "EnumValue1" : "1",
        "EnumValue2" : "2"
    }
}

// Schema Compliant
"EnumValue0"
"EnumValue1"
"EnumValue2"

// NOT Schema Compliant
"NotAnEnumValue"
```

Manage IoT device commands and events

Device commands provide the ability to remotely manage a physical device ensuring complete control over the device in addition to performing critical security, software, and hardware updates. With a large fleet of devices, knowing when a device performs a command provides oversight over your entire device implementation. A device command or an automatic update will trigger a device state change, which in turn will create a new device event. This device event will trigger a notification automatically sent to a customer-managed destination.

Topics

- Device commands
- Device Events

Device commands

A command request is the command being sent to the device. A command request includes a payload that specifies the action to be taken such as turning on the light bulb. To send a device command, the `SendManagedThingCommand` API is called on behalf of the end user by Managed Integrations and the command request is sent to the device.

The response to a `SendManagedThingCommand` is a `traceId` and you can use this `traceId` to track the command delivery and any related command response workflows wherever possible.

For more information on the `SendManagedThingCommand` API operation, see SendManagedThingCommand.

`UpdateState` action

To update the state of a device such as the time a light turns on, use the `UpdateState` action when calling the `SendManagedThingCommand` API. Provide the data model property and new value you are updating in `parameters`. The below example illustrates a `SendManagedThingCommand` API request updating the `OnTime` of a light bulb to 5.

```
{
  "Endpoints": [
    {
      "endpointId": "1",
      "capabilities": [
```

```
        {
          "id": "matter.OnOff",
          "name": "On/Off",
          "version": "1",
          "actions": [
            {
              "name": "UpdateState",
              "parameters": {
                "OnTime": 5
              }
            }
          ]
        }
      ]
    }
  ]
}
```

`ReadState` action

To get the latest state of a device including the current values of all data model properties, use the `ReadState` action when calling the `SendManagedThingCommand` API. In `propertiesToRead`, you can use the following options:

- Provide a specific data model property to get the latest value on such as `OnOff` determining if a light is on or off.
- Use the wildcard operator (*) to read *all* device state properties for a capability.

The below examples illustrate both scenarios for a `SendManagedThingCommand` API request using the `ReadState` action:

```
{
  "Endpoints": [
    {
      "endpointId": "1",
      "capabilities": [
        {
          "id": "aws.OnOff",
          "name": "On/Off",
          "version": "1",
          "actions": [
```

```
            {
              "name": "ReadState",
              "parameters": {
                "propertiesToRead": [ "OnOff" ]
              }
            }
          ]
        }
      ]
    }
  ]
}
```

```
{
  "Endpoints": [
    {
      "endpointId": "1",
      "capabilities": [
        {
          "id": "aws.OnOff",
          "name": "On/Off",
          "version": "1",
          "actions": [
            {
              "name": "ReadState",
              "parameters": {
                "propertiesToRead": [ "*" ]
              }
            }
          ]
        }
      ]
    }
  ]}
```

Device Events

A device event includes the current state of the device. This can mean the device has changed state, or is reporting its state even if the state has not changed. It includes property reports and events that are defined in the data model. An event could be a washing machine cycle has completed or thermostat has reached the targeted temperature set by the end user.

Device event notifications

An end user can subscribe to specific customer-managed destinations that they create for updates on specific device events. To create a customer-managed destination, call the `CreateDestination` API. When a device event is reported to Managed Integrations by the device, the customer-managed destination is notified if one exists.

Tagging your managed integrations resources

To help you manage and organize your resources, you can optionally assign your own metadata to each of these resources in the form of tags. This section describes tags and shows you how to create them.

Tag basics

You can use tags to categorize your managed integrations resources in different ways (for example, by purpose, owner, or environment). This is useful when you have many resources of the same type — you can quickly identify a resource based on the tags you've assigned to it. Each tag consists of a key and optional value, both of which you define. For example, you can define a set of tags for your thing types that helps you track devices by type. We recommend that you create a set of tag keys that meets your needs for each kind of resource. Using a consistent set of tag keys makes it easier for you to manage your resources.

You can search for and filter resources based on the tags you add or apply. You can also use tags to control access to your resources as described in Using tags with IAM policies.

For ease of use, the Tag Editor in the AWS Management Console provides a central, unified way to create and manage your tags. For more information, see Working with Tag Editor in Working with the AWS Management Console.

You can also work with tags using the AWS CLI and the managed integrations API. You can associate tags with managed things, provisioning profiles, credential lockers, and over-the-air (OTA) tasks when you create them by using the `Tags` field in the following commands:

- CreateManagedThing
- CreateProvisioningProfile
- CreateCredentialLocker
- CreateOtaTask
- CreateAccountAssociation

You can add, modify, or delete tags for existing resources that support tagging by using the following commands:

- TagResource

- ListTagsForResource
- UntagResource

You can edit tag keys and values, and you can remove tags from a resource at any time. You can set the value of a tag to an empty string, but you can't set the value of a tag to null. If you add a tag that has the same key as an existing tag on that resource, the new value overwrites the old value. If you delete a resource, any tags associated with the resource are also deleted.

Tag restrictions and limitations

The following basic restrictions apply to tags:

- Maximum number of tags per resource — 50
- Maximum key length — 127 Unicode characters in UTF-8
- Maximum value length — 255 Unicode characters in UTF-8
- Tag keys and values are case sensitive.
- Do not use the `aws:` prefix in your tag names or values. It's reserved for AWS use. You can't edit or delete tag names or values with this prefix. Tags with this prefix don't count against your tags per resource limit.
- If your tagging schema is used across multiple services and resources, remember that other services might have restrictions on allowed characters. Allowed characters include letters, spaces, and numbers representable in UTF-8, and the following special characters: + - = . _ : / @.

Using tags with IAM policies

You can apply tag-based resource-level permissions in the IAM policies you use for managed integrations API actions. This gives you better control over what resources a user can create, modify, or use. You use the `Condition` element (also called the `Condition` block) with the following condition context keys and values in an IAM policy to control user access (permissions) based on a resource's tags:

- Use `aws:ResourceTag/`*`tag-key`*`: `*`tag-value`* to allow or deny user actions on resources with specific tags.
- Use `aws:RequestTag/`*`tag-key`*`: `*`tag-value`* to require that a specific tag be used (or not used) when making an API request to create or modify a resource that allows tags.

- Use `aws:TagKeys: [tag-key, ...]` to require that a specific set of tag keys be used (or not used) when making an API request to create or modify a resource that allows tags.

> **Note**
>
> The condition context keys and values in an IAM policy apply only to those managed integrations actions where an identifier for a resource capable of being tagged is a required parameter. For example, the use of GetCustomEndpoint is not allowed or denied on the basis of condition context keys and values because no taggable resource (managed things, provisioning profiles, credential lockers, over-the-air tasks) is referenced in this request. For more information about managed integrations resources that are taggable and condition keys they support, read Actions, resources, and condition keys for AWS IoT managed integrations feature of AWS IoT Device Management.

For more information about using tags, see Controlling Access Using Tags in the *AWS Identity and Access Management User Guide*. The IAM JSON Policy Reference section of that guide has detailed syntax, descriptions, and examples of the elements, variables, and evaluation logic of JSON policies in IAM.

The following example policy applies two tag-based restrictions for the `CreateManagedThing` action. An IAM user restricted by this policy:

- Can't create a managed thing with the tag "env=prod" (in the example, see the line `"aws:RequestTag/env" : "prod"`).
- Can't modify or access a managed thing that has an existing tag "env=prod" (in the example, see the line `"aws:ResourceTag/env" : "prod"`).

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": "iotmanagedintegrations:CreateManagedThing",
```

```
      "Resource": "arn:aws:iotmanagedintegrations:us-east-1:123456789012:managed-
thing/*",
      "Condition": {
        "StringEquals": {
          "aws:RequestTag/env": "prod"
        }
      }
    },
    {
      "Effect": "Deny",
      "Action": [
        "iotmanagedintegrations:CreateManagedThing",
        "iotmanagedintegrations:DeleteManagedThing",
        "iotmanagedintegrations:GetManagedThing",
        "iotmanagedintegrations:UpdateManagedThing"
      ],
      "Resource": "arn:aws:iotmanagedintegrations:us-east-1:123456789012:managed-
thing/*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/env": "prod"
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "iotmanagedintegrations:CreateManagedThing",
        "iotmanagedintegrations:DeleteManagedThing",
        "iotmanagedintegrations:GetManagedThing",
        "iotmanagedintegrations:UpdateManagedThing"
      ],
      "Resource": "*"
    }
  ]
}
```

You can also specify multiple tag values for a given tag key by enclosing them in a list, like this:

```
        "StringEquals" : {
          "aws:ResourceTag/env" : ["dev", "test"]
```

```
        }
```

Note

If you allow or deny users access to resources based on tags, you must consider explicitly denying users the ability to add those tags to or remove them from the same resources. Otherwise, it's possible for a user to circumvent your restrictions and gain access to a resource by modifying its tags.

Managed Integrations notifications

Managed Integrations notifications deliver updates and key insights from devices. Notifications include connector events, device commands, lifecycle events, OTA (Over-the-Air) updates, and error reports. These insights provide actionable information to create automated workflows, take immediate actions, or store event data for troubleshooting.

Currently, only Amazon Kinesis data streams are supported as a destination for managed integrations notifications. You will first need to set up an Amazon Kinesis data stream and allow Managed Integrations access to the data stream before setting up notifications.

Set up Amazon Kinesis for notifications

Amazon Kinesis setup steps

- Step 1: Create an Amazon Kinesis data stream
- Step 2: Create a permissions policy
- Step 3: Navigate to the IAM dashboard and select Roles
- Step 4: Use a Custom trust policy
- Step 5: Apply your permissions policy
- Step 6: Enter a role name

To setup Amazon Kinesis for Managed Integrations notifications, follow these steps:

Step 1: Create an Amazon Kinesis data stream

An Amazon Kinesis Data Stream can ingest a large amount of data in real time, durably store the data, and make the data available for consumption by applications.

To create an Amazon Kinesis data stream

- To create a Kinesis data stream, follow the steps outlined in Create and manage Kinesis data streams.

Step 2: Create a permissions policy

Create a permissions policy that allows Managed Integrations to access your Kinesis data stream.

To create a permissions policy

- To create a permissions policy, copy the policy below and follow the steps outlined in Create policies using the JSON editor

  JSON

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

Step 3: Navigate to the IAM dashboard and select Roles

Open the IAM dashboard and click **Roles**.

To navigate to the IAM dashboard

- **Open the IAM dashboard and click Roles.**

  For more information, see IAM role creation in the *AWS Identity and Access Management* User Guide.

Step 4: Use a Custom trust policy

You can use a custom trust policy to grant Managed Integrations access to the Kinesis data stream.

To use a custom trust policy

- **Create a new role and choose Custom trust policy. Click Next.**

  The following policy allows Managed Integrations to assume the role, and the `Condition` statement helps prevent confused deputy issues.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "iotmanagedintegrations.amazonaws.com"
        },
            "Action": "sts:AssumeRole",
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": "123456789012"
            },
                "ArnLike": {
                    "aws:SourceArn": "arn:aws:iotmanagedintegrations:ca-
central-1:123456789012:*"
                }
        }
        }
    ]
}
```

Step 5: Apply your permissions policy

Add the permissions policy you created in step 2 to the role.

To add a permissions policy

- **On the Add permissions page, search for and add the permissions policy you created in step 2. Click Next.**

Step 6: Enter a role name

- **Enter a role name and click Create role.**

Set up Managed Integrations notifications

Notification setup steps

- Step 1: Give user permissions to call the CreateDestination API
- Step 2: Call the CreateDestination API
- Step 3: Call the CreateNotificationConfiguration API

To setup Managed Integrations notifications, follow these steps:

Step 1: Give user permissions to call the CreateDestination API

- **Give user permissions to call the `CreateDestination` API**

  The following policy defines the requirements for the user to call the CreateDestination API.

  See Grant a user permissions to pass a role to an AWS service in the *AWS Identity and Access Management* User Guide to get passrole permissions to managed integrations.

  JSON

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

 "Resource":"arn:aws:iam::123456789012:role/ROLE_CREATED_IN_PREVIOUS_STEP",
            "Condition":{
               "StringEquals":{

 "iam:PassedToService":"iotmanagedintegrations.amazonaws.com"
            }
         }
          },
          {
            "Effect":"Allow",
            "Action":"iotmanagedintegrations:CreateDestination",
            "Resource":"*"
```

```
        }
    ]
}
```

Step 2: Call the CreateDestination API

- **Call the `CreateDestination` API**

  After you have created your Amazon Kinesis data stream and stream access role, call the CreateDestination API to create your notification destination where the notifications will be routed to. For the `DeliveryDestinationArn` parameter, use the `arn` from your new Amazon Kinesis data stream.

```
{
    "DeliveryDestinationArn": "Your Kinesis arn"
    "DeliveryDestinationType": "KINESIS"
    "Name": "DestinationName"
    "ClientToken": "string"
    "RoleArn": "arn:aws:iam::accountID:role/ROLE_CREATED_IN_PREVIOUS_STEP"
}
```

> **Note**
>
> `ClientToken` is an idempotency token. If you retry a request that completed successfully initially using the same client token and parameters, then the retry attempt will succeed without performing any further actions.

Step 3: Call the CreateNotificationConfiguration API

- **Call the `CreateNotificationConfiguration` API**

  Lastly, use the CreateNotificationConfiguration API to create the notification configuration that routes the chosen event types to your destination represented by the Kinesis data stream. In the `DestinationName` parameter, use the same destination name as when you initially called the `CreateDestination` API.

```
{
```

```
    "EventType": "DEVICE_EVENT"
    "DestinationName" // This name has to be identical to the name in
 createDestination API
    "ClientToken": "string"
}
```

Event types monitored with Managed Integrations

The following are the event types monitored with Managed Integrations notifications:

- `DEVICE_COMMAND`
  - The status of the SendManagedThingCommand API command. Valid values are either `succeeded` or `failed`.

```
{
              "version":"0",
              "messageId":"6a7e8feb-b491-4cf7-a9f1-bf3703467718",
              "messageType":"DEVICE_COMMAND",
              "source":"aws.iotmanagedintegrations",
              "customerAccountId":"123456789012",
              "timestamp":"1731623291671",
              "region":"ca-central-1",
              "resources":[
                    "arn:aws:iotmanagedintegrations:ca-
central-1:123456789012:managed-thing/6a7e8feb-b491-4cf7-a9f1-bf3703467718"
              ],
              "payload":{
                "traceId":"1234567890abcdef0",
                "receivedAt":"2017-12-22T18:43:48Z",
                "executedAt":"2017-12-22T18:43:48Z",
                "result":"failed"
              }
}
```

- `DEVICE_COMMAND_REQUEST`
  - The command request from Web Real-Time Communication (WebRTC).

    The WebRTC standard allows communication between two peers. These peers can transmit real-time video, audio, and arbitrary data. Managed Integrations supports WebRTC to enable

these types of streaming between a customer mobile application and an end-user's device. For more information on the WebRTC standard, see WebRTC.


```
{
              "version":"0",
              "messageId":"6a7e8feb-b491-4cf7-a9f1-bf3703467718",
              "messageType":"DEVICE_COMMAND_REQUEST",
              "source":"aws.iotmanagedintegrations",
              "customerAccountId":"123456789012",
              "timestamp":"1731623291671",
              "region":"ca-central-1",
              "resources":[
                  "arn:aws:iotmanagedintegrations:ca-
central-1:123456789012:managed-thing/6a7e8feb-b491-4cf7-a9f1-bf3703467718"
              ],
              "payload":{
                "endpoints":[{
                  "endpointId":"1",
                  "capabilities":[{
                    "id":"aws.DoorLock",
                    "name":"Door Lock",
                    "version":"1.0"
                  }]
                }]
              }
}
```


- DEVICE_DISCOVERY_STATUS
  - The discovery status of the device.


```
{
      "version":"0",
      "messageId":"6a7e8feb-b491-4cf7-a9f1-bf3703467718",
      "messageType":"DEVICE_DISCOVERY_STATUS",
      "source":"aws.iotmanagedintegrations",
      "customerAccountId":"123456789012",
      "timestamp":"2017-12-22T18:43:48Z",
      "region":"ca-central-1",
      "resources":[
        "arn:aws:iotmanagedintegrations:ca-central-1:123456789012:managed-
thing/6a7e8feb-b491-4cf7-a9f1-bf3703467718"
      ],
      "payload":{
```

```
        "deviceCount": 1,
        "deviceDiscoveryId": "123",
        "status": "SUCCEEDED"
      }
}
```

- DEVICE_EVENT
  - A notification of a device event occurring.

```
{
      "version":"1.0",
      "messageId":"2ed545027bd347a2b855d28f94559940",
      "messageType":"DEVICE_EVENT",
      "source":"aws.iotmanagedintegrations",
      "customerAccountId":"123456789012",
      "timestamp":"1731630247280",
      "resources":[
        "/quit/1b15b39992f9460ba82c6c04595d1f4f"
      ],
      "payload":{
        "iotMiArrivalTimestamp":"2026-06-26T21:28:14.697Z",
        "endpoints":[{
          "endpointId":"1",
          "capabilities":[{
            "id":"aws.DoorLock",
            "name":"Door Lock",
            "version":"1.0",
            "properties":[{
              "name":"ActuatorEnabled",
              "value":"true"
            }]
          }]
        }]
      }
}
```

The DEVICE_EVENT payload contains the following fields:

- iotMiArrivalTimestamp, String (ISO 8601): The timestamp indicating when the device event was published by the device into Managed Integrations. Use this field to determine the latency between when the event was generated by the device and when Managed Integrations received it.

- `endpoints`: The list of device endpoints and their updated capability properties.

  The `timestamp` field in the outer message envelope indicates when the notification was delivered to Kinesis. The `iotMiArrivalTimestamp` in the payload indicates when the event arrived into Managed Integrations. These two timestamps can differ if there is processing latency between event arrival and the notification delivery.

- `DEVICE_LIFE_CYCLE`

  Reflects changes in status of device life cycle (this includes onboarding status and connected/disconnected status).

  - Onboarding status update event.

```
{
      "version": "1.0.0",
      "messageId": "a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6",
      "messageType": "DEVICE_LIFE_CYCLE",
      "source": "aws.iotmanagedintegrations",
      "customerAccountId": "123456789012",
      "timestamp": "2024-11-14T19:55:57.568284645Z",
      "region": "ca-central-1",
      "resources": [
        "arn:aws:iotmanagedintegrations:ca-central-1:123456789012:managed-thing/
a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6"
        ],
      "payload": {
        "deviceDetails": {
          "id": "a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6",
          "arn": "arn:aws:iotmanagedintegrations:ca-central-1:123456789012:managed-
thing/a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6",
          "createdAt": "2024-11-14T19:55:57.515841147Z",
          "updatedAt": "2024-11-14T19:55:57.515841559Z"
        },
        "status": "UNCLAIMED"
      }
}
```

  - Device connected status event.

```
{
    "version": "1.0",
    "messageId": "a1b2c3d4-e5f6-g7h8-i9j0-k1l2m3n4o5p6",
```

```
    "messageType": "DEVICE_LIFE_CYCLE",
    "source": "aws.iotmanagedintegrations",
    "customerAccountId": "123456789012",
    "timestamp": "2024-11-14T19:55:57.568284645Z",
    "region": "ca-central-1",
    "resources": [
        "arn:aws:iotmanagedintegrations:ca-central-1:123456789012:managed-thing/
a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6"
    ],
    "payload": {
        "managedThingId": "a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6",
        "managedThingArn": "arn:aws:iotmanagedintegrations:ca-
central-1:123456789012:managed-thing/a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6",
        "clientId": "iotmi-example-client-id",
        "timestamp": "1768000475344",
        "eventType": "connected",
        "sessionIdentifier": "q1w2e3r4-t5y6-u7i8-o9p0-a1s2d3f4g5h6",
        "principalIdentifier":
 "z1x2c3v4b5n6m7a8s9d0f1g2h3j4k5l6p7o8i9u0y1t2r3e4w5q6a7z8x9c0v1b2",
        "ipAddress": "192.0.2.100",
        "versionNumber": "0"
    }
}
```


- Device disconnected status event.


```
{
    "version": "1.0",
    "messageId": "b2n3m4a5-s6d7-f8g9-h0j1-k2l3z4x5c6v7",
    "messageType": "DEVICE_LIFE_CYCLE",
    "source": "aws.iotmanagedintegrations",
    "customerAccountId": "123456789012",
    "timestamp": "2024-11-14T19:55:57.568284645Z",
    "region": "ca-central-1",
    "resources": [
        "arn:aws:iotmanagedintegrations:ca-central-1:123456789012:managed-thing/
a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6"
    ],
    "payload": {
        "managedThingId": "a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6",
        "managedThingArn": "arn:aws:iotmanagedintegrations:ca-
central-1:123456789012:managed-thing/a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6",
        "clientId": "iotmi-example-client-id",
        "timestamp": "1768000492431",
```

```
        "eventType": "disconnected",
        "sessionIdentifier": "p9o8i7u6-y5t4-r3e2-w1q0-m9n8b7v6c5x4",
        "principalIdentifier":
 "a1s2d3f4g5h6j7k8l9z0x1c2v3b4n5m6q7w8e9r0t1y2u3i4o5p6a7s8d9f0g1h2",
        "versionNumber": "0",
        "disconnectReason": "CLIENT_INITIATED_DISCONNECT"
    }
}
```

- DEVICE_OTA
  - A device OTA notification.

```
{
    "version": "1.0.0",
    "messageId": "a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6",
    "messageType": "DEVICE_OTA",
    "source": "aws.iotmanagedintegrations",
    "customerAccountId": "123456789012",
    "timestamp": "2024-11-14T19:55:57.568284645Z",
    "region": "ca-central-1",
    "resources": [
        "arn:aws:iotmanagedintegrations:ca-central-1:123456789012:managed-thing/
a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6",
        "arn:aws:iotmanagedintegrations:ca-central-1:123456789012:managed-thing/
b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7",
        "arn:aws:iotmanagedintegrations:ca-central-1:123456789012:managed-thing/
c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7r8"
    ],
    "payload": {
        "operation": "CREATE_OTA",
        "otaTaskId": "ota-job-abc123def456",
        "status": "IN_PROGRESS",
        "otaType": "ONE_TIME"
    }
}
```

- DEVICE_STATE
  - A notification when the state of a device has been updated.

```
{
      "messageType": "DEVICE_STATE",
      "source": "aws.iotmanagedintegrations",
      "customerAccountId": "123456789012",
```

```
      "timestamp": "1731623291671",
      "resources": [
        "arn:aws:iotmanagedintegrations:ca-central-1:123456789012:managed-
thing/61889008880012345678"
      ],
      "payload": {
        "addedStates": {
          "endpoints": [{
            "endpointId": "nonEndpointId",
            "capabilities": [{
              "id": "aws.OnOff",
              "name": "On/Off",
              "version": "1.0",
              "properties": [{
                "name": "OnOff",
                "value": {
                  "propertyValue": "\"onoff\"",
                  "lastChangedAt": "2024-06-11T01:38:09.000414Z"
                }
              }
            ]}
          ]}
        ]}
      }
}
```

- ACCOUNT_ASSOCIATION
  - A notification when an account association state changes to IN_PROGRESS.

```
{
    "version": "1.0.0",
    "messageId": "a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6",
    "messageType": "ACCOUNT_ASSOCIATION",
    "source": "aws.iotmanagedintegrations",
    "customerAccountId": "123456789012",
    "timestamp": "2026-01-20T23:59:34.009284802Z",
    "region": "ca-central-1",
    "resources": ["arn:aws:iotmanagedintegrations:ca-
central-1:123456789012:account-association/a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6"],
    "payload": {
        "traceId": "a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6",
        "logLevel": "INFO",
        "resourceType": "account-association",
```

```
        "resourceId": "a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6",
        "connectorDestinationId": "a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6",
        "associationArn": "arn:aws:iotmanagedintegrations:ca-
central-1:123456789012:account-association/a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6",
        "state": "ASSOCIATION_IN_PROGRESS",
        "isServiceError": false,
        "isCustomerError": false,
        "details": "AccountAssociation State is updated to IN_PROGRESS during
 StartAccountAssociationRefresh"
    }
}
```

- A notification when an account association completes successfully.

```
{
    "version": "1.0.0",
    "messageId": "b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7",
    "messageType": "ACCOUNT_ASSOCIATION",
    "source": "aws.iotmanagedintegrations",
    "customerAccountId": "123456789012",
    "timestamp": "2026-01-20T23:59:44.672304821Z",
    "region": "ca-central-1",
    "resources": ["arn:aws:iotmanagedintegrations:ca-
central-1:123456789012:account-association/b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7"],
    "payload": {
        "traceId": "b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7",
        "logLevel": "INFO",
        "resourceType": "account-association",
        "resourceId": "b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7",
        "connectorDestinationId": "b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7",
        "associationArn": "arn:aws:iotmanagedintegrations:ca-
central-1:123456789012:account-association/b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7",
        "state": "ASSOCIATION_SUCCEEDED",
        "isServiceError": false,
        "isCustomerError": false,
        "details": "AccountAssociation has completed successfully"
    }
}
```

- `CONNECTOR_ERROR_REPORT`
  - This notification is sent when a cloud-to-cloud (C2C) connector reports an error while processing a request on your behalf, such as a command or device discovery that returned a non-success response.

```
{
    "version": "1.0.0",
    "messageId": "a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6",
    "messageType": "CONNECTOR_ERROR_REPORT",
    "source": "aws.iotmanagedintegrations",
    "customerAccountId": "123456789012",
    "timestamp": "2026-01-20T23:59:34.009284802Z",
    "region": "ca-central-1",
    "resources": [
        "arn:aws:iotmanagedintegrations:ca-central-1:123456789012:account-
association/a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6"
    ],
    "payload": {
        "statusCode": 0,
        "resourceType": "account-association",
        "resourceId": [
            "a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6"
        ],
        "associationArn": [
            "arn:aws:iotmanagedintegrations:ca-central-1:123456789012:account-
association/a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6"
        ],
        "connectorDestinationId": "b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7",
        "connectorDeviceId": "connectorDeviceId11597664a",
        "operation": "AWS.SendCommand",
        "operationVersion": "1.0",
        "traceId": "f4be7a5d55d34cc=_1",
        "logLevel": "ERROR",
        "message": "AWS.SendCommand operation failed during asynchronous
 invocation of the connector: Connector response exception: responseCode=xxx,
 responseMessage=Mocking a ValidationException with error code xxx",
        "managedThingId": "c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7r8"
    }
}
```


The `managedThingId` field is included when it can be resolved for the request. Fields without a value are omitted from the payload.

Cloud-to-Cloud (C2C) connectors

A cloud-to-cloud connector allows you to create and facilitate bidirectional communication between third-party devices and AWS.

Topics

- What is a cloud-to-cloud (C2C) connector?
- What is the C2C connector catalog?
- AWS Lambda functions as C2C connectors
- Managed Integrations connector workflow
- Guidelines for using a C2C (cloud-to-cloud) connector
- Build a C2C (Cloud-to-Cloud) connector
- Use a C2C (Cloud-to-Cloud) connector

What is a cloud-to-cloud (C2C) connector?

A cloud-to-cloud connector is a pre-built software package that securely links the AWS Cloud to a third-party cloud provider's endpoint. Using the C2C connector, solution providers can leverage managed integrations for AWS IoT Device Management to control devices that are connected to third-party clouds.

Managed Integrations includes a catalog of connectors where AWS customers can view and select connectors they want to integrate with. For more information, see What is the C2C connector catalog?

Managed Integrations requires every connector be implemented as an AWS Lambda function.

What is the C2C connector catalog?

The managed integrations for AWS IoT Device Management connector catalog is a collection of C2C connectors that facilitate bidirectional communication between managed integrations for AWS IoT Device Management and a third-party cloud provider. You can view the connectors in the AWS Management Console or the AWS CLI.

To use the console to view the managed integrations connector catalog

1. Open the managed integrations console
2. In the left navigation pane, choose **Managed integrations**
3. In the left navigation pane of the managed integrations console, choose **Catalog**.

AWS Lambda functions as C2C connectors

Every C2C connector Lambda function translates and transports commands and events between Managed Integrations and the corresponding actions on third-party platforms. For more information about Lambda, see What is AWS Lambda.

For example, suppose an end user owns a smart light-bulb manufactured by a third-party OEM. With a C2C connector, an end user can issue a command to turn on or off this light through a managed integrations platform. This command will then be forwarded to the Lambda function hosted in the connector, which will translate the request into an API call against the third-party platform to turn the device on or off.

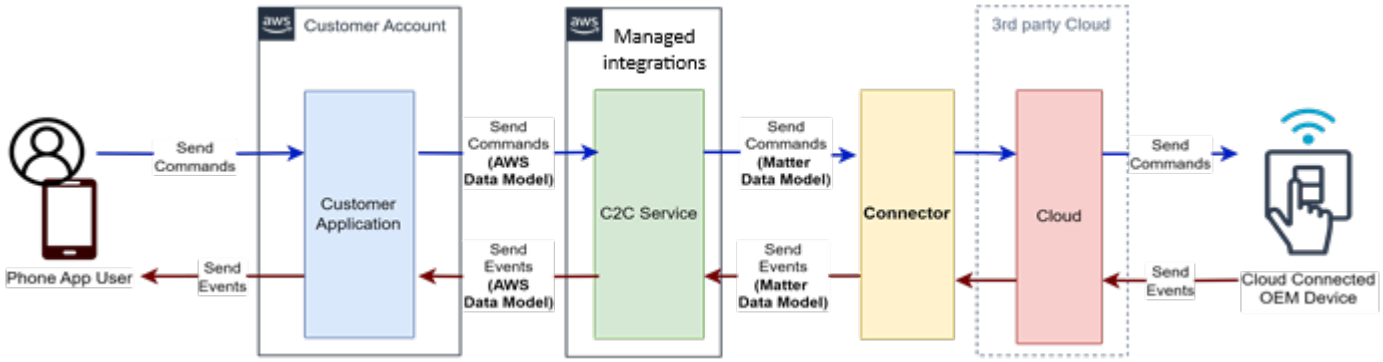
The Lambda function is required when you call the `CreateCloudConnector` API. The code that is deployed into the Lambda function must implement all of the interfaces and functionality as mentioned in Build a C2C (Cloud-to-Cloud) connector.

Managed Integrations connector workflow

Developers must register C2C connectors with managed integrations for AWS IoT Device Management. This registration process creates a logical connector resource that customers can access to use the connector.

> **Note**
>
> A C2C connector is a set of metadata created within managed integrations for AWS IoT Device Management to describe the connector.

The following diagram depicts a C2C connector's role when sending a command from the mobile application to a cloud-connected device. The C2C connector acts as a translation layer between managed integrations for AWS IoT Device Management and a third-party cloud platform.



Guidelines for using a C2C (cloud-to-cloud) connector

Any C2C connector you create is your content, and any C2C connector created by another customer that you access is third-party content. AWS does not create or manage any C2C connectors as part of managed integrations.

You may share your C2C connectors with other managed integrations customers. If you do, you authorize AWS as your service provider to list those C2C connectors and related contact information on the AWS console and you understand that other AWS customers may contact you. You are solely responsible for granting customers access to your C2C connectors and for any terms governing another AWS customer's access to your C2C connectors.

Build a C2C (Cloud-to-Cloud) connector

The following sections cover the steps to build a C2C (Cloud-to-Cloud) connector for managed integrations for AWS IoT Device Management.

Topics

- Prerequisites
- Required permissions
- C2C connector requirements
- Authorization
- Implement C2C connector interface operations
- Invoke your C2C connector
- Add permissions to your IAM Role
- Manually test your C2C connector

Prerequisites

Before you create a C2C (Cloud-to-Cloud) connector, you need the following:

- An AWS account to host your C2C connector and to register it through managed integrations. For more information, see Create an AWS account.
- When you build your connector, you need certain IAM permissions (see Required Permissions section below).
- Determine which authorization type your connector will support. Managed Integrations supports OAuth 2.0 authorization and General Authorization.

  For OAuth 2.0 Connectors:

  If your connector will support OAuth 2.0, the developer of the connector must have the following:

  - **Client ID** from the third-party cloud to associate with your C2C connector
  - **Client secret** from the third-party cloud to associate with your C2C connector
  - **OAuth 2.0 authorization URL**
  - **OAuth 2.0 token URL**

  For Custom Authorization Connectors (also referred to as General/Custom Authorization Connectors):

  If your connector supports any non-OAuth based authorization mechanism, we refer to it as General authorization connector and the connector user will require to persist the credentials for this authorization scheme in AWS Secrets Manager. The credentials for this non-OAuth authorization scheme could be tokens or API keys or other credentials, which are expected to be persisted in AWS Secrets Manager.
- Third-Party API Requirements:
  - Authorization material specific to the authorization scheme which could be API Keys/tokens (for OAuth)
  - Any allowlisting for the OAuth callback URL hosted by AWS

> **Note**
>
> Some third parties explicitly allowlist an OAuth redirect URL, while others have a workflow where users can log in and register the OAuth URL. Consult with the specific

third party to understand what is required to allowlist the managed integrations OAuth redirection endpoint.

Required permissions

When you build your connector, you need certain IAM permissions. In addition to the `iotmanagedintegrations:` permissions for the actions, you need the following permissions:

API-Specific Permissions

- CreateAccountAssociation, CreateConnectorDestination, GetAccountAssociation, and StartAccountAssociationRefresh require:
  - `secretsmanager:GetSecretValue`
- CreateCloudConnector requires:
  - `lambda:Invoke`

General Authorization Permissions

If your connector supports General Authorization, your connector Lambda execution role must also have:

- `secretsmanager:GetSecretValue`
- `kms:Decrypt`

These permissions are needed to retrieve credentials from customer AWS Secrets Manager. For more information, see Lambda permissions for GeneralAuthorization.

Additional Resources

For more information about `iotmanagedintegrations:` permissions and actions, see Actions defined by AWS Managed Integrations.

C2C connector requirements

The C2C connector you develop facilitates the bidirectional communication between managed integrations for AWS IoT Device Management and a third-party vendor cloud. The connector must

implement interfaces for managed integrations for AWS IoT Device Management to perform actions on behalf of end users.

These interfaces provide the functionality to:

- Discover end-user devices
- Initiate device commands that are sent from managed integrations for AWS IoT Device Management
- Identify users

To support the device operations, the connector must manage the translation of the request and response messages between managed integrations for AWS IoT Device Management and the related third party platform.

Core Requirements

The following are requirements for the C2C connector:

1. **OAuth 2.0 Compliance (if applicable)**

   If your connector supports OAuth 2.0, the third-party authorization server must conform to OAuth 2.0 standards as well as the configurations listed in OAuth configuration requirements.

2. **Matter Data Model Compliance**

   A C2C connector will be required to interpret identifiers from AWS implementations of the Matter Data Model and must emit the responses and events that are compliant with AWS implementations of the Matter Data Model. For more information, see AWS implementation of the Matter data model.

3. **SigV4 Authentication**

   A C2C connector must be able to call the managed integrations for AWS IoT Device Management APIs with `SigV4` authentication. For asynchronous events sent with the SendConnectorEvent API, the same AWS account credentials used to register the connector must be used to sign the related SendConnectorEvent request.

4. **Required Operations**

   The connector must implement the following four operations:

   - `AWS.ActivateUser` - Retrieve user identifier and activate user

- `AWS.DiscoverDevices` - Discover end-user devices
- `AWS.SendCommand` - Send commands to devices
- `AWS.DeactivateUser` - Deactivate user and revoke access

5. **Event Forwarding**

   When your C2C connector receives third-party events related to device command responses or device discovery, it must forward them to Managed Integrations with the `SendConnectorEvent` API. For more information on these events and the `SendConnectorEvent` API, see SendConnectorEvent.

> **Note**
>
> The `SendConnectorEvent` API is part of managed integrations SDK and is used, instead of manual building and signing of requests.

Authorization

C2C connectors can support OAuth 2.0 authorization, General Authorization, or both. The authorization type determines how your connector authenticates with the third-party platform and manages access to end user devices.

OAuth 2.0 Authorization

OAuth 2.0 provides user-level authorization through account linking. Each end user authenticates with the third-party platform and grants permission for the connector to access their devices. This ensures that device access is scoped to individual user accounts with explicit user consent.

General Authorization

General Authorization uses credentials such as API keys or tokens stored in AWS Secrets Manager. A single set of credentials can control devices across multiple end users. This approach is useful when the third-party platform doesn't support OAuth 2.0 or when you need to manage devices at scale without individual user authorization flows.

> **Note**
>
> Your connector can implement both authorization types in parallel, providing compatibility with diverse authorization frameworks.

Topics

- OAuth 2.0 requirements for account linking
- General/Custom Authorization requirements for connector developers

OAuth 2.0 requirements for account linking

Every C2C connector relies on an OAuth 2.0 authorization server to authenticate end users. Through this server, end users link their third-party accounts with the customer's device platform. Account linking is the first step required by an end user to use devices supported by your C2C connector. For more information on the different roles in account linking and OAuth 2.0, see Account linking roles.

While your C2C connector does not need to implement specific business logic to support the authorization flow, the OAuth2.0 authorization server associated with your C2C connector must meet the OAuth configuration requirements.

> **Note**
>
> Managed Integrations for AWS IoT Device Management only supports OAuth 2.0 with an authorization code flow. See RFC 6749 for more information.

Account linking is a process that allows managed integrations and the connector to access an end user's devices by using an access token. This token provides managed integrations for AWS IoT Device Management with the end user's permission, such that the connector can interact with the end user's data through API calls. For more information, see Account linking workflow.

We recommend that you don't log these sensitive tokens in any logs. If however they are stored in logs, we recommend that you use CloudWatch Logs data protection policies to mask the tokens in the logs. For more information, see Help protect sensitive log data with masking.

Managed Integrations for AWS IoT Device Management does not get an access token directly; it does so through the Authorization Code Grant Type. First, managed integrations for AWS IoT Device Management must obtain an authorization code. It then exchanges the code for an access token and refresh token. The refresh token is used to request a new access token when the old access token expires. If both the access token and refresh token are expired, you must perform the account-linking flow again. You can do this with the `StartAccountAssociationRefresh` API operation.

Important

Issued access token must be scoped per user, but not per the OAuth client. The token should not provide access to all devices of all users under the client.
The authorization server must do one of the following:

- Issue access tokens that contain extractable end-user (resource owner) ID, such as a JWT-token.
- Return the end-user ID for each issued access token.

OAuth configuration requirements

The following table illustrates the required parameters from your OAuth authorization server for managed integrations for AWS IoT Device Management to perform account linking:

OAuth Server Parameters

Field	Required	Comment
`clientId`	Yes	A public identifier for your application. It's used to initiate authorization flows and can be shared publicly.
`clientSecret`	Yes	A secret key used to authenticate the application with the authorization server, especially when exchanging an authorization code for an access token. It should

		be kept confidential and not shared publicly.
`authorizationType`	Yes	The type of authorization supported by this authorization configuration. Currently, "OAuth 2.0" is the only value supported.
`authUrl`	Yes	The authorization URL for the third-party cloud provider.
`tokenUrl`	Yes	The token URL for the third-party cloud provider.
`tokenEndpointAuthenticationScheme`	Yes	Authentication scheme of either “HTTP_BASIC” or “REQUEST_BODY_CREDENTIALS”. HTTP_BASIC signals that the client credentials are included in the authorization header, while the ladder signals they are included in the request body.

The OAuth server that you use must be configured so that access token string values must be Base64 encoded with the UTF-8 character set.

Account linking roles

To create a C2C connector, you'll need an OAuth 2.0 authorization server and account linking. For more information, see Account linking workflow.

OAuth 2.0 defines the following four roles when implementing account linking:

1. Authorization server
2. Resource owner (End User)
3. Resource server

4. Client

The following define each of these OAuth roles:

Authorization Server

The authorization server is the server that identifies and authenticates the identity of an end user in a third-party cloud. The access tokens provided by this server can link the AWS end user's customer platform account and their third-party platform account. This process is referred to as account linking.

The authorization server supports account linking by providing the following:

- Displays a login page for the end user to log in to your system. This is typically referred to as an authorization endpoint.
- Authenticates the end user in your system.
- Generates an authorization code that identifies the end user.
- Passes the authorization code to managed integrations for AWS IoT Device Management.
- Accepts the authorization code from managed integrations for AWS IoT Device Management and returns an access token that managed integrations for AWS IoT Device Management can use to access the end user's data in your system. This is typically completed through a separate URI, called a token URI or endpoint.

Important

The authorization server must support the OAuth 2.0 Authorization Code flow to be used with an managed integrations for AWS IoT Device Management Connector. managed integrations for AWS IoT Device Management also supports the authorization code flow with Proof Key for Code Exchange (PKCE).
The authorization server must either:

- Issue access tokens that contain extractable end-user or resource owner ID, for example JWT-tokens
- Be able to return the end-user ID for each issued access token

Otherwise, your connector will not be able to support the required `AWS.ActivateUser`operation. This will prevent connector usage with Managed Integrations.

If the connector developer or owner does not maintain their own authorization server, the authorization server used must provide authorization for resources managed by the connector developers third party platform. This means that any tokens received by Managed Integrations from the authorization server must provide meaningful security boundaries on devices (the resource). For example, an end users token does not allow for commands on another end users device; the permissions provided by the token are mapped to resources within the platform. Consider the *Lights Incorporated* example. When an end user starts the account linking flow with their connector, they are redirected to the *Lights Incorporated* login page which fronts their authorization server. Once they have logged in and granted permissions to the client, they provide a token that gives the connector access to resources within their *Lights Incorporated* account.

Resource owner (End User)

As the resource owner, you allow an managed integrations for AWS IoT Device Management customer access to resources associated with your account by performing account linking. For example, consider the smart bulb an end user has onboarded to the *Lights Incorporated* mobile application. The resource owner refers to the end user account that has purchased and onboarded the device. In our example the resource owner is modeled as a *Lights Incorporated* OAuth2.0 account. As the resource owner, this account provides permissions to issue commands and manage the device.

Resource server

This is the server that hosts protected resources which require authorization to access (device data). The AWS customer needs to access to protected resources on behalf of an end user, and they do so through managed integrations for AWS IoT Device Management connectors post account linking. Considering the smart bulb from before as an example, the resource server is a cloud-based service owned by *Lights Incorporated* that manages the bulb after it has been onboarded. Through the resource server, the resource owner can issue commands to the smart bulb, such as turning it on and off. The protected resource only provides permissions to the end user's account and other accounts/entities they may have provided permission to.

Client

In this context, the client is your C2C connector. A client is defined as an application that is granted access to resources within a resource server on behalf of the end user. The account linking process represents the connector, the client, requesting access to an end user's resources within the third-party cloud.

Although the connector is the OAuth client, managed integrations for AWS IoT Device Management performs operations on behalf of the connector. For example, managed integrations for AWS IoT Device Management makes requests to the authorization server to get an access token. The connector is still considered the client because it is the only component that ever accesses the protected resource (device data) in the resource server.

Consider the smart bulb that has been onboarded by an end user. After account linking has been completed between the customer platform and the *Lights Incorporated* authorization server, the connector itself will communicate with the resource server to retrieve information about the end user's smart bulb. The connector can then receive commands from the end user. This includes turning the light on or off on their behalf through the *Lights Incorporated* resource server. Thus, we designate the connector as the client.

Account linking workflow

For a customer's managed integrations for AWS IoT Device Management platform to interact with an end-user’s devices on your third-party platform through your C2C connector, it obtains the access token through the following workflow:

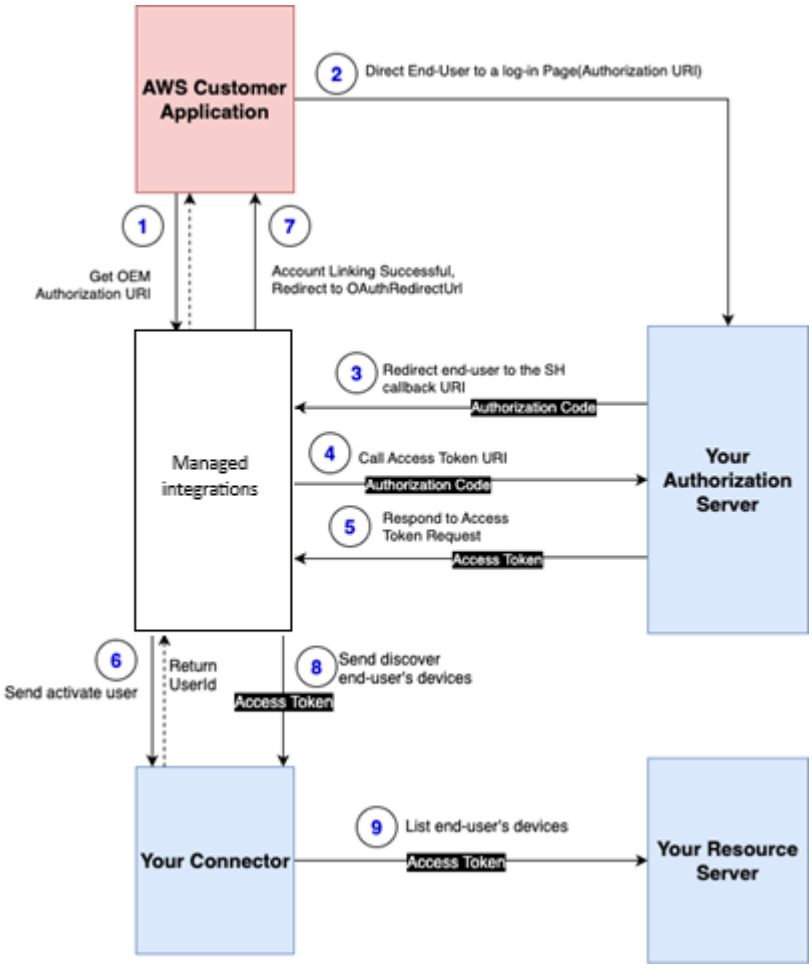
1. When a user initiates the onboarding of third-party devices through the customer application, managed integrations for AWS IoT Device Management returns Authorization URI as well as the AssociationId.
2. The application front-end stores the AssociationId and redirects the end user to the login page of the third-party platform.
    - The end user signs in. The end user grants the client access to their device data.
3. The third-party platform creates an authorization code. The end user is redirected to managed integrations for AWS IoT Device Management platform callback URI including the code attached to the redirect request.
4. Managed Integrations exchanges this code with the third-party platform token URI.
5. The token URI validates the authorization code and returns an OAuth2.0 access token and refresh token, associated with the end user.
6. Managed Integrations calls the C2C connector with `AWS.ActivateUser` operation to complete the Account Linking flow and get UserId.
7. Managed Integrations returns OAuthRedirectUrl (from the Connector Policy configuration) of the successful authentication page to the customer application.

> **Note**
>
> In case of failures, managed integrations for AWS IoT Device Management appends error and error_description query parameters to the URL providing error details to the customer application.

8. The customer application redirects the end user to the OAuthRedirectUrl. At this point the application front-end knows AssociationId of the association from the first step.

   All subsequent requests made from managed integrations for AWS IoT Device Management through the C2C connector to the third-party cloud platform, such as commands to discover devices and send commands, will include the OAuth2.0 access token.

The following diagram shows the relationship between key components of account linking:



General/Custom Authorization requirements for connector developers

General Authorization enables your connector to use credentials (such as API keys, tokens, or username/password combinations) instead of OAuth 2.0 user tokens. Unlike OAuth 2.0, which provides user-level authorization through account linking, General Authorization allows a single set of credentials to control devices across multiple end users.

> **Note**
>
> Throughout this documentation, Custom Authorization is referred to as General Authorization. Both terms describe the same authorization mechanism. In the following sections, we use "General Authorization" for consistency.

This section explains how to implement General Authorization support in your connector AWS Lambda function. If you are a customer configuring General Authorization for an existing connector, see General/Custom Authorization requirements.

Topics

- What is General Authorization?
- How General Authorization uses AWS Secrets Manager
- General Authorization request format
- General Authorization workflow
- Lambda permissions for GeneralAuthorization

What is General Authorization?

General Authorization is any non-OAuth authorization mechanism that allows your connector to authorize with third-party platforms using customer credentials. With General Authorization, Managed Integrations delegates credential management to your connector, and a single set of credentials can control devices across multiple end users.

This is useful for scenarios where you have a business relationship with the device vendor and need to manage devices at scale without individual user authorization flows.

When to use General Authorization

Consider implementing General Authorization support in your connector when:

- The third-party platform does not support OAuth 2.0
- The third-party platform provides custom authorization material such as API keys or credentials that can reside in AWS Secrets Manager
- You need to manage devices at scale without individual user authorization flows

> **Note**
>
> Your connector can implement both authorization types in parallel, providing compatibility with diverse authorization frameworks.

How General Authorization uses AWS Secrets Manager

AWS Secrets Manager is a secret storage service that protects sensitive credentials such as API keys and tokens. Secrets are encrypted using AWS Key Management Service keys. For more information, see the AWS Secrets Manager User Guide.

For General Authorization, customers store authorization credentials in Secrets Manager, and grant your C2C connector permission to access these secrets. When managed integrations invokes your connector, it provides the Secrets Manager ARN and version ID in the request header. Your connector retrieves the secret value and uses it to authorize with the third-party platform.

This approach ensures that managed integrations never handles long-term credentials directly. Your connector maintains full control over credential management and token generation, making the solution extensible to any authorization mechanism supported by your third-party platform.

> **Important**
>
> Managed Integrations does not access or manage the credentials stored in the customer's AWS Secrets Manager. Your connector has full control over credential retrieval, parsing, and usage.

> **Important**
>
> We recommend that you don't log sensitive credentials or tokens in any logs. If however they are stored in logs, we recommend that you use CloudWatch Logs data protection

policies to mask the tokens in the logs. For more information, see Help protect sensitive log data with masking.

General Authorization request format

When Managed Integrations invokes your connector for a General Authorization account association, the request header contains a AWS Secrets Manager reference instead of an OAuth token. The request structure is consistent across all connector operations (`AWS.ActivateUser`, `AWS.DiscoverDevices`, `AWS.SendCommand`, and `AWS.DeactivateUser`).

Example: General Authorization Request

```
{
  "header": {
    "auth": {
      "secretsManager": {
        "arn": "arn:aws:secretsmanager:us-east-1:123456789012:secret:my-api-key-
AbCdEf",
        "versionId": "a1b2c3d4-5678-90ab-cdef-1234567890ab"
      },
      "type": "GeneralAuthorization"
    }
  },
  "payload": {
    "operationName": "AWS.DiscoverDevices",
    "operationVersion": "1.0",
    "connectorId": "Your-Connector-Id",
    ...
  }
}
```

Example: OAuth 2.0 Request (for comparison)

```
{
  "header": {
    "auth": {
      "token": "ashriu32yr97feqy7afsaf",
      "type": "OAuth2.0"
    }
  },
```

```
  "payload": {
    "operationName": "AWS.DiscoverDevices",
    "operationVersion": "1.0",
    "connectorId": "Your-Connector-Id",
    ...
  }
}
```

> **Note**
>
> Your connector must handle both request formats. Check the `auth.type` field to determine which authorization method to use for each request.

General Authorization workflow

When your connector receives a General Authorization request, follow this workflow:

- **Check authorization type** - Check the `auth.type` field in the request header to determine if the request uses General Authorization
- **Extract Secrets Manager reference** - Extract the AWS Secrets Manager ARN and version ID from the `auth.secretsManager` object
- **Retrieve secret** - Call the AWS Secrets Manager `GetSecretValue` API using the provided ARN and version ID
- **Parse credentials** - Parse the secret value to extract authorization credentials (the format depends on your third-party platform's requirements)
- **Generate tokens (if needed)** - If needed, use the credentials to generate an access token or perform additional authorization steps required by the third-party platform
- **Authorize API calls** - Use the credentials or generated token to authorize API calls to the third-party platform
- **Process operation** - Process the connector operation (`AWS.DiscoverDevices`, `AWS.SendCommand`, etc.) using the authorized connection

> **Note**
>
> Your connector is responsible for all credential management, including token generation, refresh, and error handling. Managed Integrations only provides the reference to the secret; it does not manage the credentials themselves.

Lambda permissions for GeneralAuthorization

Your connector Lambda execution role must have permission to retrieve secrets from the customer's AWS Secrets Manager. Add the following permissions to your Lambda execution role policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "secretsmanager:GetSecretValue"
      ],
      "Resource": "arn:aws:secretsmanager:*:*:secret:*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "kms:Decrypt"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "kms:ViaService": "secretsmanager.*.amazonaws.com"
        }
      }
    }
  ]
}
```

Permission Explanation

- `secretsmanager:GetSecretValue` - Allows your Lambda to retrieve secret values

- `kms:Decrypt` - Required because secrets are encrypted using AWS Key Management Service keys

> **Note**
>
> The example policy allows access to any secret. In production, you should restrict the `Resource` field to only the secrets your connector needs. However, since customers create their own secrets, you may need to use a wildcard or document the naming convention customers should follow.

The customer will also grant your Lambda permission to access their specific secret through the secret's resource policy.

Implement C2C connector interface operations

Managed Integrations for AWS IoT Device Management defines four operations your AWS Lambda must handle to qualify as a connector. Your C2C connector must implement each of the following operations:

1. `AWS.ActivateUser` - Managed Integrations for AWS IoT Device Management service calls this API to retrieve a globally unique user identifier. For OAuth 2.0, this is associated with the provided OAuth 2.0 token. This operation can optionally be used to perform any additional requirements for the account linking process.
2. `AWS.DiscoverDevices` - Managed Integrations for AWS IoT Device Management service calls this API to your connector for discovering user's devices
3. `AWS.SendCommand` - Managed Integrations for AWS IoT Device Management service calls this API to your connector for sending commands for user's devices
4. `AWS.DeactivateUser` - Managed Integrations for AWS IoT Device Management service calls this API to your connector for deactivating user's access token to delink in your authorization server.

Invocation Details

Managed Integrations for AWS IoT Device Management always invokes the Lambda function with a JSON string payload through the AWS Lambda `invokeFunction` action. Request operations must include an `operationName` field in every request payload.

Invocation Settings:

- **Timeout:** 2 seconds per invocation
- **Retries:** 5 retry attempts on failure

Implementation Example

The Lambda you implement for your connector will parse an `operationName` from the request payload and implement the corresponding functionality to map to the third-party cloud:

```
public ConnectorResponse handleRequest(final ConnectorRequest request)
        throws OperationFailedException {
    Operation operation;
    try {
        operation = Operation.valueOf(request.payload().operationName());
    } catch (IllegalArgumentException ex) {
        throw new ValidationException(
           "Unknown operation '%s'".formatted(request.payload().operationName()),
           ex
        );
    }

    return switch (operation) {
        case ActivateUser -> activateUserManager.activateUser(request);
        case DiscoverDevices -> deviceDiscoveryManager.listDevices(request);
        case SendCommand -> sendCommandManager.sendCommand(request);
        case DeactivateUser -> deactivateUser.deactivateUser(request);
    };
}
```

> **Note**
>
> The developer of the connector must implement the `activateUserManager.activateUser(request)`,

`deviceDiscoveryManager.listDevices(request)`, `sendCommandManager.sendCommand(request)`, and `deactivateUser.deactivateUser` operations listed in the preceding example.

Request Format Examples

The following examples detail generic connector requests from Managed Integrations, in which common fields to every required interface are present. From the examples, you can see there is both a request header and request payload. Request headers are common throughout every operation interface.

OAuth 2.0 example:

```
{
        "header": {
                "auth": {
                        "token": "ashriu32yr97feqy7afsaf",
                        "type": "OAuth2.0"
                }
        },
        "payload":{
                "operationName": "AWS.SendCommand",
                "operationVersion": "1.0",
                "connectorId": "exampleId",
        …
        }
}
```

General Authorization example:

```
{
        "header": {
                "auth": {
                        "secretsManager": {
                                "arn": "string",
                                "versionId": "string"
                        },
                        "type": "GeneralAuthorization"
                }
        },
```

```
        "payload":{
                "operationName": "AWS.SendCommand",
                "operationVersion": "1.0",
                "connectorId": "exampleId",
        …
        }
}
```

Default request headers

The default header fields vary depending on the authorization type. Your connector must handle both OAuth 2.0 and General Authorization request headers.

OAuth 2.0 default header:

```
{
    "header": {
        "auth": {
            "token": string,    // End user's Access Token
            "type": "OAuth2.0"
        }
    }
}
```

General Authorization default header:

```
{
    "header": {
        "auth": {
            "secretsManager": {
                "arn": "string",
                "versionId": "string"
            },
            "type": "GeneralAuthorization"
        }
    }
}
```

Header Parameters

Field	Required/Optional	Description

`header:auth`	Yes	Authorization information provided by the C2C connector builder during their connector registration.
`header:auth:token`	Conditional	Authorization token of user generated by the third-party cloud provider and linked to `connectorAssociationID` . Required for OAuth 2.0, not present for General Authorization.
`header:auth:secretsManager`	Conditional	AWS Secrets Manager ARN and version ID containing authorization credentials. Required for General Authorization, not present for OAuth 2.0.
`header:auth:type`	Yes	The type of authorization: `OAuth2.0` or `GeneralAuthorization` .

Note

All requests to your connector will include authorization information. For OAuth 2.0, this includes the end user's access token. For General Authorization, this includes the AWS Secrets Manager ARN and version ID. You can assume that the appropriate authorization has already been established.

Request Payload

In addition to common headers, every request will have a payload. While this payload will have unique fields for every operation type, each payload has a set of default fields that will always be present.

Request payload fields:

- `operationName`: The operation of a given request, equal to one of the following values: `AWS.ActivateUser`, `AWS.SendCommand`, `AWS.DiscoverDevices`, `AWS.DeactivateUser`.
- `operationVersion`: Every operation is versioned to allow its evolution over time and providing stable interface definition for third-party connectors. Managed Integrations passes a version field in the payload of all requests.
- `connectorId`: The ID of the connector in which the request has been sent to.

Default Response Headers

Every operation will respond with an ACK to managed integrations for AWS IoT Device Management that confirms your C2C connector has received the request and begun to process it.

Example Generic Response Example

```
{
  "header":{
   "responseCode": 200
  },
  "payload":{
   "responseMessage": “Example response!”
  }
}
```

Example Response Header Format

```
{
    "header": {
        "responseCode": Integer
    }
}
```

Response Header Field

Default response header and field

Field	Required/Optional	Comment

`header:responseCode`	Yes	ENUM of values that indicate the execution status of the request.

Throughout the various Connector Interfaces and API schemas described in this document there is a `responseMessage` or `Message` field. This is an optional field used for the C2C connector Lambda to respond with any context regarding the request and its execution. Preferably, any errors resulting in a status code other than `200` should include a message value describing the error.

Respond to C2C connector operation requests with the SendConnectorEvent API

Managed Integrations for AWS IoT Device Management expects your connector to behave asynchronously for every `AWS.SendCommand` and `AWS.DiscoverDevices` operation. This means that the initial response to these operations, simply "acknowledges" that your C2C connector has received the request.

Using the `SendConnectorEvent` API, your connector is expected to send the event types from the list below for `AWS.DiscoverDevices` and `AWS.SendCommand` operations, as well as proactive device events (such as a light being manually turned on and off).

Example Workflow

If your C2C connector receives a `DiscoverDevices` request, Managed Integrations for AWS IoT Device Management expects it to:

- Respond synchronously with the response format defined above
- Invoke the `SendConnectorEvent` API with a DEVICE_DISCOVERY event

The `SendConnectorEvent` API call can take place anywhere you have access to your C2C connector Lambda AWS account credentials. The device discovery flow is not successful until managed integrations for AWS IoT Device Management receives this event.

> **Note**
>
> Alternatively, the `SendConnectorEvent` API call can occur before the C2C connector Lambda invocation response if necessary. However, this flow contradicts the asynchronous model for software development.

SendConnectorEvent API

Your connector calls this managed integrations for AWS IoT Device Management API to send device events. Only 3 types of events are accepted:

- **"DEVICE_DISCOVERY"** - Used to send list of discovered devices within third-party cloud for a specific access token
- **"DEVICE_COMMAND_RESPONSE"** - Used to send a specific device event as a result of command execution
- **"DEVICE_EVENT"** - Used for any event that originates from the device which is not the direct result of a user-based command. This can serve as a general event type to proactively report device state changes or notifications

Implement the AWS.ActivateUser operation

The `AWS.ActivateUser` operation is required for Managed Integrations for AWS IoT Device Management to retrieve a user identifier from an end user. For OAuth 2.0, Managed Integrations for AWS IoT Device Management will pass the OAuth token within the request header. For General Authorization, Managed Integrations for AWS IoT Device Management will pass the AWS Secrets Manager reference. Your connector must include the globally unique user identifier in the response payload.

Requirements

The following list outlines the requirements for your connector to facilitate a successful `AWS.ActivateUser` flow:

- Your C2C connector Lambda can process an `AWS.ActivateUser` operation request message from Managed Integrations for AWS IoT Device Management.
- Your C2C connector Lambda can determine a unique user identifier. For OAuth 2.0, this can be extracted from the token itself if it's a JWT token, or requested from the authorization server. For General Authorization, this may be retrieved from your third-party platform or derived from the authorization context.

`AWS.ActivateUser` Workflow

Step 1: Managed Integrations Invokes Your Lambda

Managed Integrations for AWS IoT Device Management invokes your C2C connector Lambda with one of the following payloads, depending on the authorization type:

OAuth 2.0 Request:

```
{
   "header": {
        "auth": {
            "token": "ashriu32yr97feqy7afsaf",
            "type": "OAuth2.0"
        }
   },
   "payload": {
        "operationName": "AWS.ActivateUser",
        "operationVersion": "1.0.0",
        "connectorId": "Your-Connector-ID"
   }
}
```

General Authorization request:

```
{
   "header": {
        "auth": {
            "secretsManager": {
                "arn": "string",
                "versionId": "string"
            },
            "type": "GeneralAuthorization"
        }
   },
   "payload": {
        "operationName": "AWS.ActivateUser",
        "operationVersion": "1.0.0",
        "connectorId": "Your-Connector-ID"
   }
}
```

Step 2: Determine User ID

The C2C connector determines the user ID to include in the `AWS.ActivateUser` response.

- **For OAuth 2.0:** This is retrieved from the token or by querying your authorization server.

- **For General Authorization:** This may be retrieved from your third-party platform or derived from the authorization context.

Step 3: Respond with User Identifier

The C2C connector responds to `AWS.ActivateUser` operation Lambda invocation, including the default payload as well as the corresponding user identifier within the `userId` field.

Response Format:

```
{
     "header": {
          "responseCode":200
     },
     "payload": {
          "responseMessage": "Successfully activated user with connector-id `Your-Connector-Id.”,
          "userId": "123456"
     }
}
```

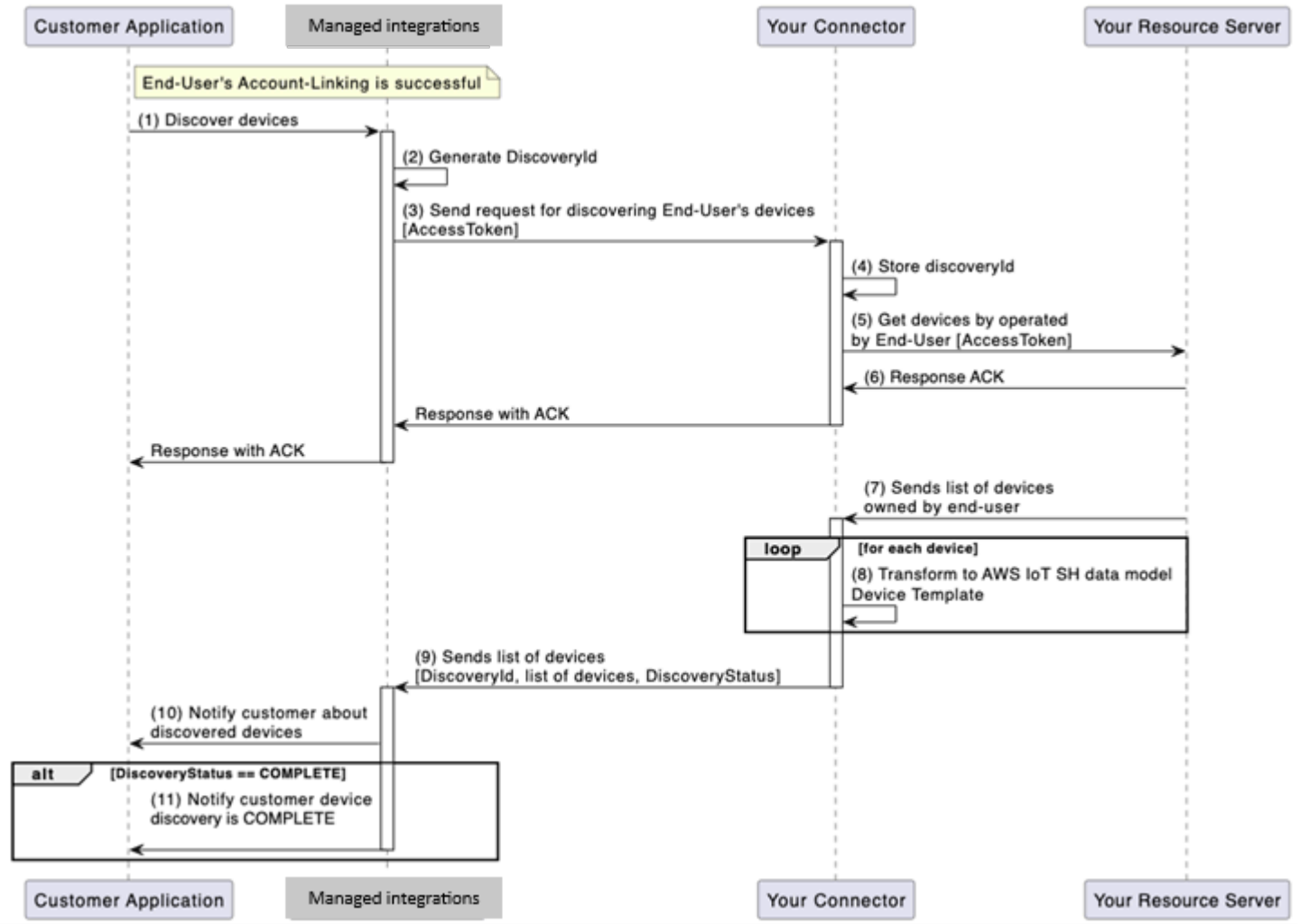
Implement the AWS.DiscoverDevices operation

Device discovery aligns the list of physical devices owned by the end user with the digital representations of those end user devices maintained in Managed Integrations for AWS IoT Device Management. It is performed by an AWS customer on devices owned by the end user. For OAuth 2.0, this occurs after account linking is completed. For General Authorization, this can occur after the account association is created.

Device discovery is an asynchronous process where Managed Integrations for AWS IoT Device Management calls a connector to initiate the device discovery request. A C2C connector returns a list of discovered end user devices asynchronously with a reference identifier (referred to as `deviceDiscoveryId`) generated by Managed Integrations for AWS IoT Device Management.

`AWS.DiscoverDevices` Workflow

The following diagram illustrates the device discovery workflow between the end user and Managed Integrations for AWS IoT Device Management:



Workflow Steps

1. **Customer initiates device discovery** - The customer initiates the device discovery process on behalf of the end user.
2. **Managed integrations generates reference ID** - Managed Integrations for AWS IoT Device Management generates a reference identifier called `deviceDiscoveryId` for the device discovery request generated by the AWS Customer.
3. **Device discovery request sent** - Managed Integrations for AWS IoT Device Management sends a device discovery request to the C2C connector using the `AWS.DiscoverDevices` operation interface, including authorization information (OAuth access token or AWS Secrets Manager reference) as well as the `deviceDiscoveryId`.
4. **Connector stores deviceDiscoveryId** - Your connector stores `deviceDiscoveryId` to be included in the `DEVICE_DISCOVERY` event. This event will also contain a list of discovered end

user's devices, and must be sent to Managed Integrations for AWS IoT Device Management with the SendConnectorEvent API as a DEVICE_DISCOVERY event.

5. **Connector calls resource server** - Your C2C connector shall call resource server to fetch all the devices owned by the end user.
6. **Connector responds with ACK** - Your C2C connector Lambda responds to the Lambda invocation (invokeFunction) with the ACK response back to Managed Integrations for AWS IoT Device Management, acting as the initial response for the AWS.DiscoverDevices operation. Managed Integrations for AWS IoT Device Management notifies customer with an ACK to their initiated device discovery process.
7. **Resource server returns device list** - Your resource server sends you a list of devices owned and operated by the end user.
8. **Convert device format** - Your connector converts each end user device into the Managed Integrations for AWS IoT Device Management required device format, including ConnectorDeviceId, ConnectorDeviceName and Capability report for each device.
9. **Provide UserId** - The C2C connector also provides UserId of the discovered devices owner. It may be retrieved from your resource server either as part of the device list or in a separate call depending on your resource server implementation.
10. **Call SendConnectorEvent API** - Next, your C2C connector will call the Managed Integrations for AWS IoT Device Management API, SendConnectorEvent, over SigV4 using AWS account credentials and with operation parameter set as "DEVICE_DISCOVERY". Each device in the list of devices sent to Managed Integrations for AWS IoT Device Management will be represented by device-specific parameters such as connectorDeviceId, connectorDeviceName, and a capabilityReport.
    - Based on your resource server response, you need to notify Managed Integrations for AWS IoT Device Management accordingly.
    - For example, if your resource server has paginated response to list of discovered devices for an end user, then for each poll you can send an individual DEVICE_DISCOVERY operation event, with a statusCode parameter of 3xx. If your device discovery is still in process, then repeat steps 5, 6, and 7.
11. **Managed integrations notifies customer** - Managed Integrations for AWS IoT Device Management sends a notification to customer about discovered the end user's devices.
12. **Completion notification** - If your C2C connector sends a DEVICE_DISCOVERY operation event with the statusCode parameter updated with a value of 200, then Managed Integrations for AWS IoT Device Management will notify customer of the device discovery workflow completion.

> **Important**
>
> Steps 7 through 11 can occur before step 6, if desired. For example, if your third-party platform has an API to list an end user's devices, the DEVICE_DISCOVERY event can be sent with `SendConnectorEvent` before the C2C connector Lambda responds with the typical ACK.

C2C connector Requirements for Device Discovery

The following list outlines the requirements for your C2C connector to facilitate a successful device discovery:

- The C2C connector Lambda can process a device discovery request message from Managed Integrations for AWS IoT Device Management and handle the `AWS.DiscoverDevices` operation.
- Your C2C connector can call the Managed Integrations for AWS IoT Device Management APIs via SigV4 using the credentials of the AWS account used for registering the connector.

Device Discovery Process

Step 1: Managed Integrations Triggers Device Discovery

Send a POST request to `DiscoverDevices` with one of the following JSON payloads, depending on the authorization type:

OAuth 2.0 Request:

```
/DiscoverDevices
{
   "header": {
        "auth": {
            "token": "ashriu32yr97feqy7afsaf",
            "type": "OAuth2.0"
        }
   },
   "payload": {
        "operationName": "AWS.DiscoverDevices",
        "operationVersion": "1.0",
        "connectorId": "Your-Connector-Id",
```

```
        "deviceDiscoveryId": "12345678",
        "connectorDeviceIdList": []
    }
}
```

> **Note**
>
> The `connectorDeviceIdList` parameter is an optional filter that allows you to specify a list of device IDs to discover. When empty (`[]`), all devices associated with the account will be discovered. When populated with specific device IDs, only those devices will be included in the discovery response.

General Authorization request:

```
/DiscoverDevices
{
   "header": {
        "auth": {
            "secretsManager": {
                "arn": "string",
                "versionId": "string"
            },
            "type": "GeneralAuthorization"
        }
   },
   "payload": {
        "operationName": "AWS.DiscoverDevices",
        "operationVersion": "1.0",
        "connectorId": "Your-Connector-Id",
        "deviceDiscoveryId": "12345678",
        "connectorDeviceIdList": []
   }
}
```

> **Note**
>
> The `connectorDeviceIdList` parameter is an optional filter that allows you to specify a list of device IDs to discover. When empty (`[]`), all devices associated with the account will

be discovered. When populated with specific device IDs, only those devices will be included in the discovery response.

General Authorization Request:

```
/DiscoverDevices
{
   "header": {
        "auth": {
            "secretsManager": {
                "arn": "string",
                "versionId": "string"
            },
            "type": "GeneralAuthorization"
        }
   },
   "payload": {
        "operationName": "AWS.DiscoverDevices",
        "operationVersion": "1.0",
        "connectorId": "Your-Connector-Id",
        "deviceDiscoveryId": "12345678",
        "connectorDeviceIdList": []
   }
}
```

Note

The `connectorDeviceIdList` parameter is an optional filter that allows you to specify a list of device IDs to discover. When empty (`[]`), all devices associated with the account will be discovered. When populated with specific device IDs, only those devices will be included in the discovery response.

Step 2: Connector Acknowledges Discovery

The connector sends an acknowledgment with the following JSON response:

```
{
     "header": {
          "responseCode":200
```

```
    },
    "payload": {
         "responseMessage": "Discovering devices for discovery-job-id '12345678' with
 connector-id `Your-Connector-Id`"
    }
}
```

Step 3: Connector Sends Device Discovery Event

Send a POST request to `/connector-event/{your_connector_id}` with the following JSON payload:

```
AWS API - /SendConnectorEvent
URI – POST /connector-event/{your_connector_id}
{
   "UserId": "6109342",
   "Operation": "DEVICE_DISCOVERY",
   "OperationVersion": "1.0",
   "StatusCode": 200,
   "DeviceDiscoveryId": "12345678",
   "ConnectorId": "Your_connector_Id",
   "Message": "Device discovery for discovery-job-id '12345678' successful",
   "Devices": [
        {
            "ConnectorDeviceId": "Your_Device_Id_1",
            "ConnectorDeviceName": "Your-Device-Name",
            "CapabilityReport": {
          "nodeId":"1",
    "version":"1.0.0",
     "endpoints":[{
     "id":"1",
     "deviceTypes":["Camera"],
     "clusters":[{
      "id":"0x0006",
      "revision":1,
      "attributes":[{
       "id":"0x0000",
      }],
      "commands":["0x00","0x01"],
      "events":["0x00"]
     }]
    }]
   }
```

```
        }
    ]
}
```

Populate Brand and Model for discovered devices

Your C2C connector can optionally include a `DeviceMetadata` object for each device in the `DEVICE_DISCOVERY` event to report the device's brand and model. These values are then returned in the ListDiscoveredDevices and ListManagedThings responses.

Include a `DeviceMetadata` object in each device entry and map:

- `DeviceMetadata.Brand` (String) → `Brand`
- `DeviceMetadata.Model` (String) → `Model`

The following example shows a device entry with `DeviceMetadata` included:

```
{
    "ConnectorDeviceId": "Your_Device_Id_1",
    "ConnectorDeviceName": "Your-Device-Name",
    "CapabilityReport": { ... },
    "DeviceMetadata": {
        "Brand": "ExampleBrand",
        "Model": "ExampleModel-X1"
    }
}
```

Both keys are case-sensitive strings. If you omit `DeviceMetadata`, omit an individual key, or provide a non-string value, the corresponding field is not available when the device is retrieved.

Construct a CapabilityReport for the DISCOVER_DEVICES event

As seen in the event structure defined above, every device reported in a DISCOVER_DEVICES event, serving as response to an `AWS.DiscoverDevices` operation, will require a CapbilityReport to describe the corresponding device's capabilities. A `CapabilityReport` tells managed integrations for AWS IoT Device Management device capabilities in a Matter compliant format.

Required Fields in CapabilityReport

- `nodeId`, String: Identifier for the devices node containing the following `endpoints`

- `version`, String: Version of this device node, set by the connector developer
- `endpoints`, List<Cluster>: List of AWS implementations of the Matter Data Model supported by this device endpoint.
  - `id`, String: Endpoint identifier set by connector developer
  - `deviceTypes`, List<String>: List of device types this endpoint captures, i.e. "Camera".
  - `clusters`, List<Cluster>: List of AWS implementations of the Matter Data Model this endpoint supports.
    - `id`, String: Cluster identifier as defined by the Matter standard.
    - `revision`, Integer: Cluster revision number as defined by the Matter standard.
    - `attributes`, Map<String, Object>: Map of attribute identifiers and their corresponding current device state values, with identifiers and valid values defined by the matter standard.
      - `id`, String: Attribute ID as defined by AWS implementations of the Matter Data Model.
      - `value`, Object: The current value of the attribute defined by the attribute ID. The type of 'value' can change depending on the attribute. The `value` field is optional for each attribute and should only be included if your connector lambda can determine the current state during discovery.
    - `commands`, List<String>: List of command IDs supported this cluster as defined by the Matter standard.
    - `events`, List<String>: List of event IDs supported this cluster as defined by the Matter standard.

For the current list of supported capabilities and their corresponding AWS implementations of the Matter Data Model refer to the latest release of the Data Model documentation.

Implement the AWS.SendCommand operation

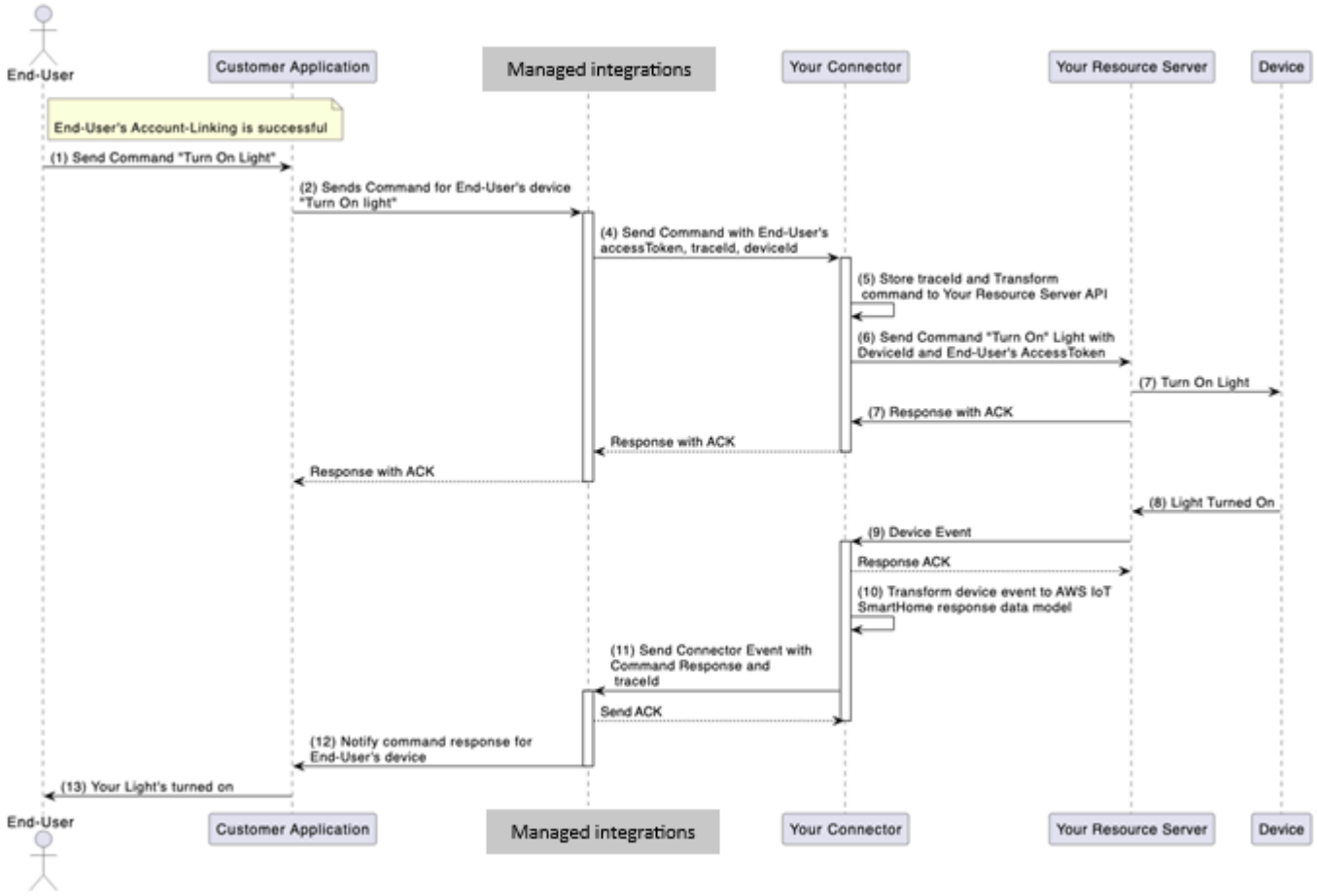
The `AWS.SendCommand` operation allows Managed Integrations for AWS IoT Device Management to send commands initiated by the end user via the AWS customer to your resource server. Your resource server may support multiple types of devices, where each type has its own response model.

Command execution is an asynchronous process where Managed Integrations for AWS IoT Device Management sends a request for command execution with a `traceId`, which your connector will include in a command response sent back to Managed Integrations for AWS IoT Device Management via the `SendConnectorEvent` API. Managed Integrations for AWS IoT Device

Management expects the resource server to return a response acknowledging that command was received, but not necessarily indicating that command was executed.

Device Command Execution Workflow

The following diagram illustrates the command execution flow with an example where the end user tries to turn on the lights of their house:



Workflow Steps

1. **End user sends command** - An end user sends a command to turn on a light using the AWS customer's application.
2. **Customer relays command** - The customer relays the command information to Managed Integrations for AWS IoT Device Management with the end user's device information.
3. **Managed integrations generates traceId** - Managed Integrations for AWS IoT Device Management generates "traceId" that your connector will use while sending command responses back to service.

4. **Command request sent to connector** - Managed Integrations for AWS IoT Device Management sends the command request to your connector, using the `AWS.SendCommand` operation interface.
   - The payload defined by this interface consists of the device identifier, device commands formulated as Matter endpoints/clusters/commands, the end user's access token, and other required parameters.
5. **Connector stores traceId** - Your connector stores the `traceId` to be included in the command response.
   - Your connector translates Managed Integrations for AWS IoT Device Management command request into your resource server's appropriate format.
6. **Connector gets UserId** - Your connector gets `UserId` from the provided end user's access token and associates it with the command.
   - The `UserId` may be either retrieved from your resource server using a separate call or extracted from the access token in case of JWT and similar tokens.
   - Implementation depends on your resource server and access token details.
7. **Connector calls resource server** - Your connector calls the resource server to "Turn On" end user's light.
8. **Resource server interacts with device** - The resource server interacts with the device.
   - The connector relays to Managed Integrations for AWS IoT Device Management that the resource server has delivered the command, responding with an ACK as the initial, synchronous command response.
   - Managed Integrations for AWS IoT Device Management then relays it back to customer application.
9. **Device executes command** - After the device turns on the light, that device event is captured by your resource server.
10. **Resource server sends device event** - Your resource server sends the device event to connector.
11. **Connector transforms event** - Your connector transforms the device event generated by resource server into Managed Integrations for AWS IoT Device Management DEVICE_COMMAND_RESPONSE event operation type.
12. **Connector calls SendConnectorEvent** - Your connector calls the `SendConnectorEvent` API with operation as "DEVICE_COMMAND_RESPONSE".
    - It attaches the `traceId` provided by Managed Integrations for AWS IoT Device Management in the initial request.

13. **Managed integrations notifies customer** - Managed Integrations for AWS IoT Device Management notifies the customer about end user's device state change.
14. **Customer notifies end user** - Customer notifies the end user that the device's light has turned on.

> **Note**
>
> Your resource server configuration determines the logic for handling failed device command request and response messages. This includes message retry attempts using the same referenceId for the command.

C2C connector Requirements for Device Command Execution

The following list outlines the requirements for your C2C connector to facilitate a successful device command execution.

- The C2C connector Lambda can process `AWS.SendCommand` operation request messages from managed integrations for AWS IoT Device Management.
- Your C2C connector must keep track of commands sent to your resource server and map it with appropriate `traceId`.
- You can call managed integrations for AWS IoT Device Management service API's via SigV4 using AWS credentials of AWS account used for registering the C2C connector.

Command Execution Process

Step 1: Managed Integrations Sends Command to Connector

Send a POST request with one of the following payloads, depending on the authorization type:

OAuth 2.0 Request:

```
/Send-Command
{
     "header": {
          "auth": {
              "token": "ashriu32yr97feqy7afsaf",
              "type": "OAuth2.0"
```

```
          }
     },
     "payload": {
          "operationName": "AWS.SendCommand",
          "operationVersion": "1.0",
          "connectorId": "Your-Connector-Id",
          "connectorDeviceId": "Your_Device_Id",
          "traceId": "traceId-3241u78123419",
          "endpoints": [{
              "id": "1",
              "clusters": [{
                  "id": "0x0202",
                  "commands": [{
                      "0xff01":
                          {
                              "0x0000": "3”
                    }
                  }]
              }]
          }]
     }
  }
```

General Authorization request:

```
/Send-Command
{
     "header": {
          "auth": {
              "secretsManager": {
                  "arn": "string",
                  "versionId": "string"
              },
              "type": "GeneralAuthorization"
          }
     },
     "payload": {
          "operationName": "AWS.SendCommand",
          "operationVersion": "1.0",
          "connectorId": "Your-Connector-Id",
          "connectorDeviceId": "Your_Device_Id",
          "traceId": "traceId-3241u78123419",
          "endpoints": [{
```

```
            "id": "1",
            "clusters": [{
                "id": "0x0202",
                "commands": [{
                    "0xff01":
                        {
                            "0x0000": "3"
                  }
                }]
            }]
        }]
    }
}
```

Step 2: C2C Connector ACK Command

```
{
     "header":{
          "responseCode":200
     },
     "payload":{
          "responseMessage": "Successfully received send-command request for connector
 'Your-Connector-Id' and connector-device-id 'Your_Device_Id'"
     }
  }
```

Step 3: Connector Sends Device Command Response Event

```
AWS-API: /SendConnectorEvent
URI: POST /connector-event/{Your-Connector-Id}

{
   "UserId": "End-User-Id",
   "Operation": "DEVICE_COMMAND_RESPONSE",
   "OperationVersion": "1.0",
   "StatusCode": 200,
   "Message": “Example message”,
   "ConnectorDeviceId": "Your_Device_Id",
   "TraceId": "traceId-3241u78123419",
   "MatterEndpoint": {
        "id": "1",
        "clusters": [{
            "id": "0x0202",
```

```
            "attributes": [
                {
                    "0x0000": “3”
                }
            ],
            "commands": [
                "0xff01":
                {
                    "0x0000": "3”
                }
   ]
        }]
    }
}
```

Note

Device state changes as a result of a command execution will not be reflected in Managed Integrations for AWS IoT Device Management until the corresponding DEVICE_COMMAND_RESPONSE event has been received through the SendConnectorEvent API. This means that until Managed Integrations for AWS IoT Device Management receives the event from prior step 3, regardless of whether or not your connector invocation response denotes success, the device state will not be updated.

Important

Do not include attributes in the DEVICE_COMMAND_RESPONSE payload unless the device has confirmed that the state change was actually applied. A DEVICE_COMMAND_RESPONSE without attributes serves as an acknowledgment that the command was dispatched to the third party, and results in a DEVICE_COMMAND notification. To report that attribute values have been updated on the device, send a separate DEVICE_EVENT with the updated attributes. This distinction prevents false positives where a command appears to succeed but the device never received the state change, for example, when a device was recently disconnected.

Interpreting matter 'endpoints' included in AWS.SendCommand request

Managed Integrations will use the device capabilities reported during device discovery to determine what commands a device can accept. Every device capability is modeled through AWS implementations of the Matter Data Model; thus, all incoming commands will be derived from the `commands` field within a given cluster. It is your connector's responsibility to parse the `endpoints` field, determining the corresponding Matter command, and translating it such that correct command reaches the device. Typically, this means translating the Matter data model into the related API requests.

After the command has been executed, your connector then determines which `attributes` defined by the AWS implementations of the Matter Data Model have changed as a result. These changes are then reported to managed integrations for AWS IoT Device Management via API DEVICE_COMMAND_RESPONSE events sent with the `SendConnectorEvent` API.

Consider the `endpoints` field included in the following example `AWS.SendCommand` payload:

```
        "endpoints": [{
            "id": "1",
            "clusters": [{
                "id": "0x0202",
                "commands": [{
                    "0xff01":
                        {
                            "0x0000": "3”
                  }
                }]
            }]
        }]
```

From this object, the connector can determine the following:

1. Set the endpoint and cluster information:

    a. Set the endpoint `id` to "1".

> **Note**
>
> If a device defines multiple endpoints such that a single cluster (such as On/Off) can control multiple capabilities (i.e. turn a light on/off as well as turning a strobe on/off), this id is used to route the command to the correct capability.

   b. Set the cluster `id` to "0x0202" (Fan Control cluster).

2. Set the command information:

   a. Set the command identifier to "0xff01" (Update State command defined by AWS).

   b. Update the included attribute identifiers with the values provided in the request.

3. Update the attribute:

   a. Set the attribute identifier to "0x0000" (FanMode attribute of the Fan Control Cluster).

   b. Set the attribute value to "3" (High fan speed).

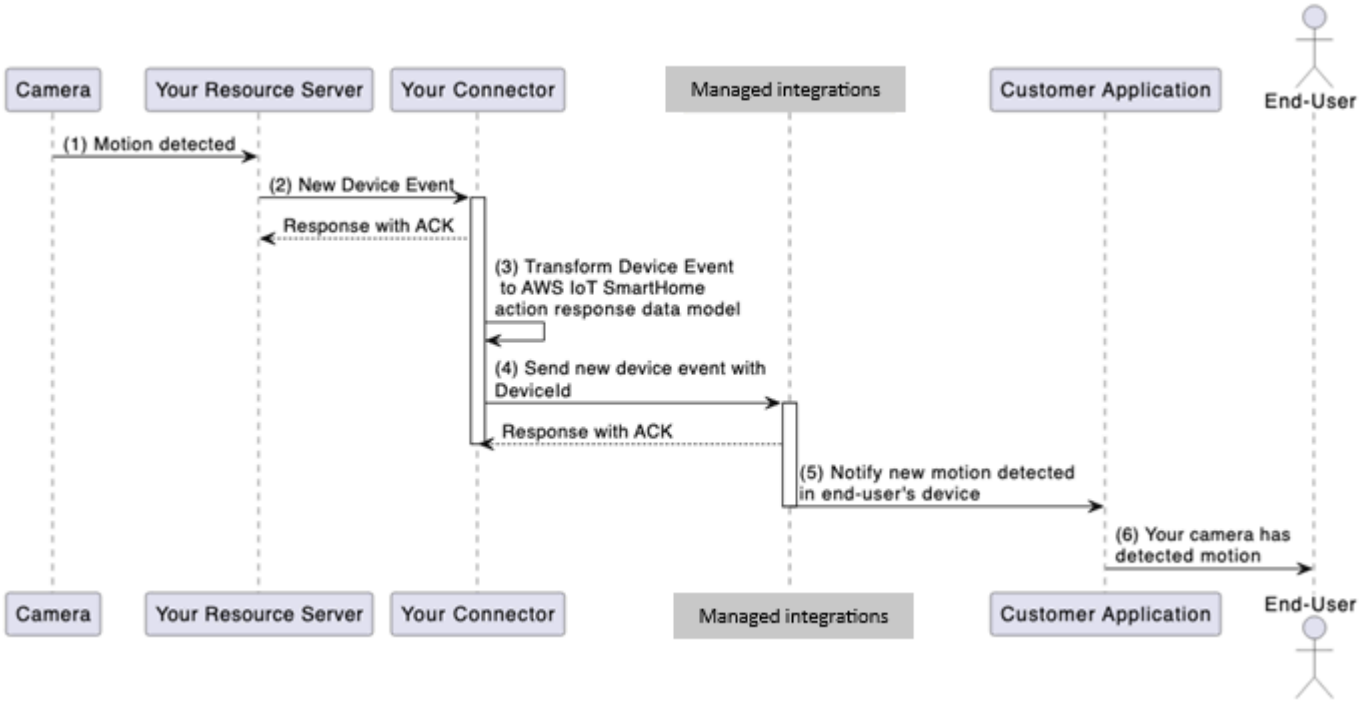
Managed Integrations has defined two “custom” command types that are not strictly defined by AWS implementations of the Matter Data Model: The ReadState and UpdateState commands. To get and set Matter defined cluster attributes, Managed Integrations will send your connector an `AWS.SendCommand` request with command IDs pertaining to UpdateState (id: 0xff01) or ReadState (id: 0xff02), with corresponding parameters of attributes that must be either updated or read. These commands can be invoked for ANY device type for attributes that are set as mutable (updatable) or retrievable (readable) from the corresponding AWS implementation of the Matter Data Model.

Send device events with the SendConnectorEvent API

Device initiated events overview

While the `SendConnectorEvent` API is used to asynchronously respond to `AWS.SendCommand` and `AWS.DiscoverDevices` operations, it is also used to notify managed integrations of any device-initiated events. Device initiated events can be defined as any event generated by a device without a user initiated command. These device events may include, but are not limited to, device state changes, motion detection, battery levels, and more. You can send these events back to managed integrations using the `SendConnectorEvent` API with operation **DEVICE_EVENT**.

The following section uses a smart camera installed at a home as an example to further explain the working flow of these events:



Device event workflow

1. Your camera detects motion for which it generates an event that is sent to your resource server.
2. Your resource server processes the event and sends it to your C2C connector.
3. Your connector translates this event to the managed integrations for AWS IoT Device Management `DEVICE_EVENT` interface.
4. Your C2C connector sends this device event to Managed Integrations using the `SendConnectorEvent` API with Operation set to "DEVICE_EVENT".
5. Managed Integrations identifies the relevant customer, and relays this event back to the customer.
6. The customer receives this event and displays it to user via user identifier.

For more information on the `SendConnectorEvent` API operation, see `SendConnectorEvent` in the managed integrations for AWS IoT Device Management API Reference Guide.

Device initiated events requirements

The following are some requirements for device-initiated events.

- Your C2C connector resource should be able to receive asynchronous device events from your resource server
- Your C2C connector resource should be able to call managed integrations for AWS IoT Device Management service API's via SigV4 using AWS credentials of the AWS account used for registering the C2C connector.

The following example demonstrates a connector sending a device-originated event via the SendConnectorEvent API:

```
AWS-API: /SendConnectorEvent
URI: POST /connector-event/{Your-Connector-Id}

{
   "UserId": "Your-End-User-ID",
   "Operation": "DEVICE_EVENT",
   "OperationVersion": "1.0",
   "StatusCode": 200,
   "Message": None,
   "ConnectorDeviceId": "Your_Device_Id",
   "MatterEndpoint": {
        "id": "1",
        "clusters": [{
            "id": "0x0202",
            "attributes": [
                {
                    "0x0000": "3"
                }
            ]
        }]
    }]
}
```

From the following example, we see the following:

- This is coming from the device endpoint with ID equal to 1.
- The device capability this event relates to has a cluster ID of **0x0202**, pertaining to the Fan Control matter cluster.

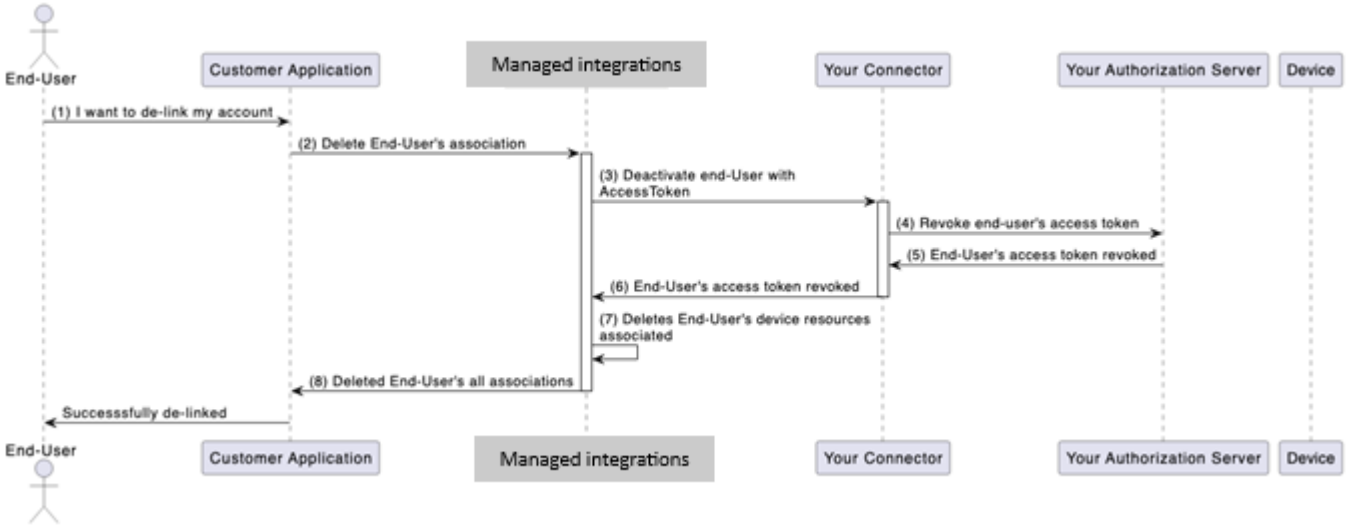
- The attribute that has changed has the ID of **0x000**, pertaining to the Fan Mode Enum within the cluster. It has updated to value **3**, pertaining to the value of **High**.
- Since `connectorId` is a parameter returned by the cloud service on creation, Connectors must query using GetCloudConnector and filter by `lambdaARN`. The lambda's own `ARN` is queried using `Lambda.get_function_url_config` API. This allows the `CloudConnectorId` to be dynamically accessed in the lambda, and not statically configured as earlier.

Implement the AWS.DeactivateUser operation

User deactivation overview

Deactivation of provided user access tokens is required when a customer deletes their AWS customer account; or when an end user would like to unlink their account in the system from AWS customer's system. In either use-case managed integrations needs to facilitate this workflow using the C2C connector.

The image below illustrates the delinking an end user account from the system



User deactivation workflow

1. User initiates delinking process between AWS customer's account and the third-party authorization server associated with the C2C connector.
2. Customer initiates deletion of user's association through managed integrations for AWS IoT Device Management.
3. Managed Integrations initiates the deactivation process via request to your connector using the `AWS.DeactivateUser` operation interface.

    - For OAuth 2.0, the user's access token is included in the header. For General Authorization, the AWS Secrets Manager reference is included in the header.
4. Your C2C connector accepts the request and performs the necessary deactivation steps. For OAuth 2.0, this includes invoking your authorization server to revoke the token and any access it provides. For General Authorization, this may include cleanup operations or notifying your third-party platform.

    - For example, events from an unlinked user account should no longer be sent to Managed Integrations after performing `AWS.DeactivateUser`.
5. Your authorization server or third-party platform processes the deactivation and sends a response back to your C2C connector.
6. Your C2C connector sends managed integrations for AWS IoT Device Management an ACK that the deactivation has been processed.
7. Managed Integrations deletes all resources owned by the end user which were associated with your resource server.
8. Managed Integrations sends an ACK to the customer, stating all associations relating to your system are deleted.
9. The customer notifies the end user that their account has been unlinked from your platform.

`AWS.DeactivateUser` Requirements

The following requirements must be met for your C2C connector to successfully handle user deactivation:

- **Request Handling:** The C2C connector Lambda function receives a request message from managed integrations to handle the `AWS.DeactivateUser` operation.
- **Token Revocation / Cleanup:**
    - **For OAuth 2.0:** The C2C connector must revoke the provided OAuth 2.0 token and the corresponding refresh token of the user within your authorization server.
    - **For General Authorization:** The C2C connector should perform any necessary cleanup or deactivation steps required by your third-party platform.

Request Examples

OAuth 2.0 Request:

The following is an example `AWS.DeactivateUser` request that your connector will receive for OAuth 2.0:

```
{
     "header": {
         "auth": {
             "token": "ashriu32yr97feqy7afsaf",
             "type": "OAuth2.0"
         }
     },
     "payload":{
         "operationName": "AWS.DeactivateUser",
         "operationVersion": "1.0",
         "connectorId": "Your-connector-Id"
     }
}
```

General Authorization example:

```
{
     "header": {
         "auth": {
             "secretsManager": {
                 "arn": "arn:aws:secretsmanager:us-east-1:123456789012:secret:my-
secret-abc123",
                 "versionId": "EXAMPLE1-90ab-cdef-fedc-ba987EXAMPLE"
             },
             "type": "GeneralAuthorization"
         }
     },
     "payload":{
         "operationName": "AWS.DeactivateUser",
         "operationVersion": "1.0",
         "connectorId": "Your-connector-Id"
     }
}
```

Invoke your C2C connector

AWS Lambda allows for resource-based policies to authorize who can invoke a Lambda. As managed integrations for AWS IoT Device Management is an AWS service, you must allow Managed Integrations to invoke your C2C connector Lambda via the resource policy.

Attach a resource policy with at least the following minimal permissions to your C2C connector Lambda. This provides Managed Integrations with Lambda function invoke privileges. This policy includes a `Condition` key to help you limit the usability of your `connectorId` to only intended users.

JSON

```
{
  "Version":"2012-10-17",
  "Id": "default",
  "Statement": [
    {
      "Sid": "Your-Desired-Policy-ID",
      "Effect": "Allow",
      "Principal": {
        "Service": "iotmanagedintegrations.amazonaws.com"
      },
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:ca-central-1:123456789012:function:connector-
lambda-name",
      "Condition": {
        "StringEquals": {
          "aws:SourceArn": "arn:aws:iotmanagedintegrations:ca-
central-1:123456789012:account-association/account-association-id"
        }
      }
    }
  ]
}
```

Add permissions to your IAM Role

All Managed Integrations APIs require AWS sigV4 authentication to invoke. SigV4 is signing protocol to authenticate AWS API requests using your AWS account credentials. The IAM role you use to invoke the Managed Integrations APIs must have the following permissions to be able to successfully invoke the APIs:

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

```
    "Sid": "Statement1",
    "Effect": "Allow",
    "Action": [
     "iotmanagedintegrations:Your-Required-Actions"
    ],
    "Resource": [
     "Your-Resource"
    ]
   }]
}
```

For additional information on adding these permissions, contact Support.

Additional resources

To register your C2C connector, you will need the following:

- The Lambda ARN designating the connector you would like to register.

Manually test your C2C connector

To manually test your C2C connector end-to-end, you must simulate both the customer and the end user.

You will need the following resources:

- An AWS Lambda ARN designating the connector you would like to test.
- A testing OAuth 2.0 user account from your cloud platform.
- A connector registered with managed integrations for AWS IoT Device Management. For more information, see Use a C2C (Cloud-to-Cloud) connector.

Use a C2C (Cloud-to-Cloud) connector

A C2C connector manages the translation of request and response messages, and enables communication between managed integrations and a third-party vendor cloud. It facilitates unified control across different device types, platforms and protocols enabling third-party devices to be onboarded and managed.

The following procedure lists the steps to use the C2C connector.

Steps to use the C2C connector:

1. **CreateCloudConnector**

   Configure a connector to enable bidirectional communication between your managed integrations and third-party vendor clouds.

   When setting up the connector, provide the following details:

   - **Name**: Choose a descriptive name for the connector.
   - **Description**: Provide a brief summary of the connector's purpose and capabilities.
   - **AWS Lambda ARN**: Specify the Amazon Resource Name (ARN) of the AWS Lambda function that will power the connector.

   Build and deploy an AWS Lambda function that communicates with third-party vendor APIs to create a connector. Next, call the CreateCloudConnector API within managed integrations, and provide the AWS Lambda function ARN for registration. Ensure that the AWS Lambda function is deployed in the same AWS account where you crete the connector in managed integrations. You will be assigned a unique **Connector ID** to identify the integration.

   Sample CreateCloudConnector API Request and Response:

```
Request:

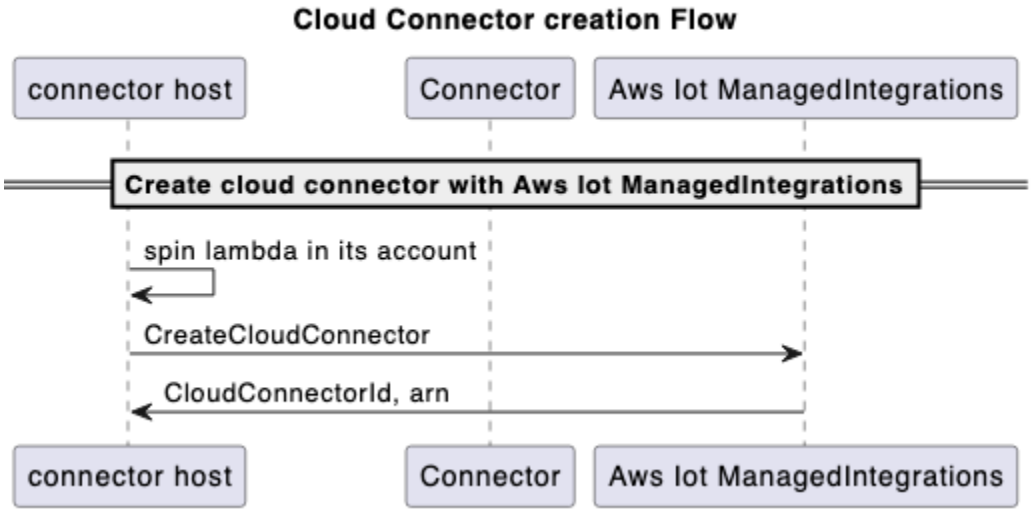
{
    "Name": "CreateCloudConnector",
    "Description": "Testing for C2C",
    "EndpointType": "LAMBDA",
    "EndpointConfig": {
        "lambda": {
            "arn": "arn:aws:lambda:us-east-1:xxxxxx:function:TestingConnector"
        }
    },
    "ClientToken": "abc"
}

Response:

{
    "Id": "string"
```

```
}
```

Creation flow:



> **Note**
>
> Use the GetCloudConnector, UpdateCloudConnector, DeleteCloudConnector, and ListCloudConnectors APIs as needed for this procedure.

2. **CreateConnectorDestination**

Configure Destinations to provide the settings and authorization credentials that connectors need to establish secure connections with third-party vendor clouds. Use Destinations to register your third-party authorization credentials with managed integrations.

Two authorization types are now supported:

- **OAuth 2.0** - For platforms using OAuth authorization (authorization URL, token URL, client credentials)
- **GeneralAuthorization** - For platforms using API keys, bearer tokens, or any non-OAuth authorization mechanism

Prerequisites

Before creating a **ConnectorDestination**, you must:

- Call the CreateCloudConnector API to create a connector. The ID that the function returns is used in the CreateConnectorDestination API API call.
- **For OAuth authorization:**
  - Retrieve the `tokenUrl` for the third-party platform (to exchange an authCode for an accessToken)
  - Retrieve the `authUrl` for the third-party platform (for end-user authorization)
  - Store the `clientId` and `clientSecret` in AWS Secrets Manager
- **For GeneralAuthorization:**
  - Store your authorization materials (API keys, bearer tokens, etc.) in AWS Secrets Manager
  - Each authorization material needs a name and Secrets Manager reference

Sample CreateConnectorDestination API Request (OAuth):

```
Request:

{
    "Name": "CreateConnectorDestination",
    "Description": "CreateConnectorDestination",
    "AuthType": "OAUTH",
    "AuthConfig": {
        "oAuth": {
            "authUrl": "https://xxxx.com/oauth2/authorize",
            "tokenUrl": "https://xxxx/oauth2/token",
            "scope": "testScope",
            "tokenEndpointAuthenticationScheme": "HTTP_BASIC",
            "oAuthCompleteRedirectUrl": "about:blank",
            "proactiveRefreshTokenRenewal": {
                "enabled": false,
                "DaysBeforeRenewal": 30
            }
        }
    },
    "CloudConnectorId": "<connectorId>",
    "SecretsManager": {
        "arn": "arn:aws:secretsmanager:*****:secret:*******",
        "versionId": "********"
    },
    "ClientToken": "***"
```

```
}

Response:

{
    "Id":"string"
}
```

Sample CreateConnectorDestination API Request (GeneralAuthorization):

```
Request:

{
    "Name": "CreateConnectorDestination",
    "Description": "GeneralAuthorization test destination",
    "AuthConfig": {
        "GeneralAuthorization": {
            "AuthMaterials": [
                {
                    "AuthMaterialName": "AuthKey1",
                    "SecretsManager": {
                        "arn": "arn:aws:secretsmanager:*****:secret:*******",
                        "versionId": "********"
                    }
                }
            ]
        }
    },
    "CloudConnectorId": "<connectorId>",
    "ClientToken": "***"
}

Response:

{
    "Id": "string"
}
```

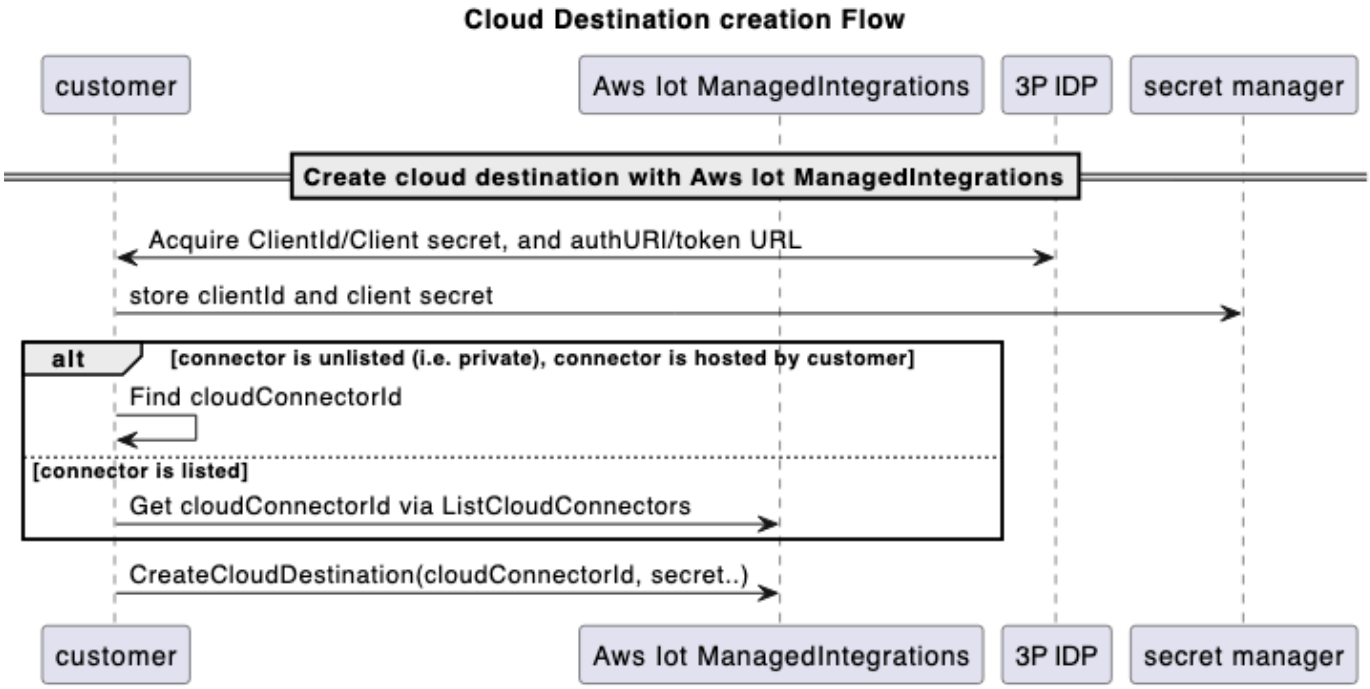
Key differences for GeneralAuthorization:

- No `AuthType` field required
- No top-level `SecretsManager` field required

- Uses `AuthConfig.GeneralAuthorization.AuthMaterials` array
- Each auth material has a name and its own Secrets Manager reference
- Supports multiple authentication materials for future use cases

Currently, ConnectorDestination also supports OAuth and GeneralAuthorization together in our ConnectorDestination.

Cloud destination creation flow:



> **Note**
>
> Use the GetConnectorDestination, UpdateConnectorDestination, DeleteConnectorDestination, and ListConnectorDestinations APIs as needed for this procedure.

3. **CreateAccountAssociation**

Associations represent the relationships between end users' third-party cloud accounts and a connector destination. After creating an Association and linking end users to managed integrations, their devices are accessible through a unique **Association ID**. This integration enables three key functions: discovering devices, sending commands, and receiving events.

Prerequisites

Before creating an **AccountAssociation** you must complete the following:

- Call the CreateConnectorDestination API to create a destination. The ID that the function returns is used in the CreateAccountAssociation API call.
- Invoke the CreateAccountAssociation API.

Sample CreateAccountAssociation API Request (OAuth):

```
Request:

{
    "Name": "CreateAccountAssociation",
    "Description": "CreateAccountAssociation",
    "ConnectorDestinationId": "<destinationId>",
    "ClientToken": "***"
}

Response:

{
    "Id":"string"
}
```

Sample CreateAccountAssociation API Request (GeneralAuthorization):

```
Request:

{
    "Name": "CreateAccountAssociation",
    "Description": "GeneralAuthorization test account association",
    "GeneralAuthorization": {
        "AuthMaterialName": "AuthKey1"
    },
    "ConnectorDestinationId": "<destinationId>",
    "ClientToken": "***"
}

Response:
```

```
{
    "AccountAssociationId": "string",
    "Arn": "string",
    "AssociationState": "ASSOCIATION_SUCCEEDED"
}
```

Key differences for GeneralAuthorization:

- Includes `GeneralAuthorization.AuthMaterialName` field
- References one of the auth materials defined in the ConnectorDestination
- No OAuth authorization URL in the response

> **Note**
>
> Use the GetAccountAssociation, UpdateAccountAssociation, DeleteAccountAssociation, and ListAccountAssociations APIs as needed for this procedure.

An **AccountAssociation** has a state that is queried from GetAccountAssociation and ListAccountAssociations APIs. These APIs show the state of the **Association**. The StartAccountAssociationRefresh API allows the refresh of an **AccountAssociation** state when its refresh token expires.

4. **Device discovery**

Each managed thing is linked to device-specific details, such as its serial number and a data model. The data model describes the device's functionality, indicating whether it's a lightbulb, switch, thermostat, or another type of device. There are two workflows for discovering third-party devices and creating managedThings: the traditional discovery flow and the pre-onboarded discovery flow.

a. **Option 1: Traditional Device Discovery Flow**

Use this workflow when you don't know the connector device IDs in advance. This flow discovers all devices associated with an account and allows you to select which devices to onboard.

i. Call StartDeviceDiscovery API to start the device discovery process.

Sample StartDeviceDiscovery API Request and Response:

```
Request:

{
    "DiscoveryType": "CLOUD",
    "AccountAssociationId": "******",
    "ClientToken": "abc"
}

Response:

{
    "Id": "string",
    "StartedAt": number
}
```

ii. Invoke GetDeviceDiscovery API to check the status of the discovery process.

iii. Invoke ListDiscoveredDevices API to list the devices discovered.

Sample ListDiscoveredDevices API Request and Response:

```
Request:

//Empty body

Response:

{
    "Items": [
    {
      "Brand": "string",
      "ConnectorDeviceId": "string",
      "ConnectorDeviceName": "string",
      "DeviceTypes": [ "string" ],
      "DiscoveredAt": number,
      "ManagedThingId": "string",
      "Model": "string",
      "Modification": "string"
    }
],
    "NextToken": "string"
```

```
}
```

iv. Invoke CreateManagedThing API to select the devices from the discovery list to be imported into managed integrations.

Sample CreateManagedThing API Request and Response:

```
Request:

{
    "Role": "DEVICE",
    "AuthenticationMaterial":
 "CLOUD:<deviceDiscoveryId>:<connectorDeviceId>",
    "AuthenticationMaterialType": "DISCOVERED_DEVICE",
    "Name": "sample-device-name",
    "ClientToken": "xxx"
}

Response:

{
   "Arn": "string", // This is the ARN of the managedThing
   "CreatedAt": number,
   "Id": "string"
}
```

v. Invoke GetManagedThing API to view this newly created `managedThing`. The status will be `UNASSOCIATED`.

vi. Invoke RegisterAccountAssociation API to associate this `managedThing` to a specific `accountAssociation`. At the end of a successful RegisterAccountAssociation API, the `managedThing` changes to `ACTIVATED` state.

Sample RegisterAccountAssociation API Request and Response:

```
Request:

{
    "AccountAssociationId": "string",
    "DeviceDiscoveryId": "string",
    "ManagedThingId": "string"
}
```

```
Response:

{
    "AccountAssociationId": "string",
    "DeviceDiscoveryId": "string",
    "ManagedThingId": "string"
}
```

b. **Option 2: Pre-onboarded Device Discovery Flow**

Use this workflow when you already know the connector device IDs before onboarding. This flow is useful for pre-provisioned devices or when you want to selectively onboard specific devices from a larger set. This approach reduces the number of API calls required to fully register and activate devices.

> **Important**
>
> To use the pre-onboarded cloud discovery flow, you must know the `connectorDeviceId` (connector device identifier) before initiating the device onboarding process. This identifier is obtained from the third-party vendor's platform or during device provisioning.

i. Invoke CreateManagedThing API with `PRE_ONBOARDED_CLOUD` authentication material type. This creates a managedThing in `PRE_ASSOCIATED` state with multiple account associations.

Sample CreateManagedThing API Request and Response (Pre-onboarded):

```
Request:

{
    "Role": "DEVICE",
    "AuthenticationMaterial":
 "CLOUD:<connectorDeviceId>:<accountAssociationId1>:<accountAssociationId2>",
    "AuthenticationMaterialType": "PRE_ONBOARDED_CLOUD",
    "Name": "pre-onboarded-device-name",
    "ClientToken": "xxx"
}

Response:
```

```
{
   "Arn": "string", // This is the ARN of the managedThing
   "CreatedAt": number,
   "Id": "string"
}
```

> **Note**
>
> The `AuthenticationMaterial` format for pre-onboarded devices is `CLOUD:<connectorDeviceId>:<accountAssociationId1>:<accountAssociat` where you can specify one or more account association IDs.

ii. (Optional) Invoke GetManagedThing API to verify the managedThing is in `PRE_ASSOCIATED` state.

iii. Call StartDeviceDiscovery API with the `connectorDeviceIdList` parameter to discover only the pre-onboarded devices.

Sample StartDeviceDiscovery API Request with connectorDeviceIdList:

```
Request:

{
    "DiscoveryType": "CLOUD",
    "AccountAssociationId": "******",
    "ConnectorDeviceIdList": [
        "connector-device-id-1",
        "connector-device-id-2",
        "connector-device-id-3"
    ],
    "ClientToken": "abc"
}

Response:

{
    "Id": "string",
    "StartedAt": number
}
```

When using `connectorDeviceIdList`, the discovery process returns only devices matching the specified connector device IDs. The connector will send a `DEVICE_DISCOVERY` event via SendConnectorEvent with the discovered device information.

iv. After the discovery completes successfully, managed integrations automatically registers the pre-onboarded managedThings to their associated account associations. The managedThing transitions from `PRE_ASSOCIATED` to `ACTIVATED` state.

> **Note**
>
> If auto-registration fails and the managedThing remains in `DISCOVERED` state, you can manually invoke the RegisterAccountAssociation API as a fallback to complete the registration process.

v. (Optional) Invoke GetManagedThing API to verify the managedThing is now in `ACTIVATED` state.

5. **Send a command to the third-party device**

To control a newly onboarded device, use the SendManagedThingCommand API, with the previously created **Association ID** and a control action based on the capability supported by the device. The connector uses stored credentials from the account linking process to authenticate with the third-party cloud and invoke the relevant API call for the operation.

> **Note**
>
> For GeneralAuthorization, the connector retrieves the authorization material (API key, bearer token, etc.) from Secrets Manager using the authorization material name specified in the AccountAssociation.

Sample SendManagedThingCommand API Request and Response:

```
Request:

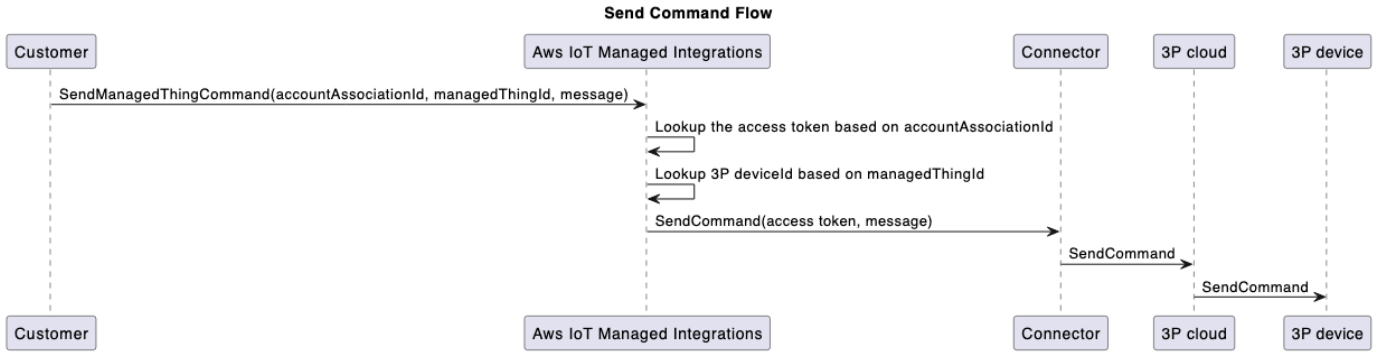
{
    "AccountAssociationId": "string",
    "ConnectorAssociationId": "string",
```

```
    "Endpoints": [
       {
          "capabilities": [
             {
                "actions": [
                   {
                      "actionTraceId": "string",
                      "name": "string",
                      "parameters": JSON value,
                      "ref": "string"
                   }
                ],
                "id": "string",
                "name": "string",
                "version": "string"
             }
          ],
          "endpointId": "string"
       }
    ]
}

Response:

{
   "TraceId": "string"
}
```

Send command to the third-party device flow:



6. **Connector sends events to managed integrations**

   The SendConnectorEvent API captures four types of events from the connector to managed integrations, represented by the following enum values for the **Operation Type** parameter:

- **DEVICE_COMMAND_RESPONSE**: The asynchronous response that the connector sends in response to a command.
- **DEVICE_DISCOVERY**: In response to a device discovery process, connector sends the list of discovered devices to managed integrations, it uses the SendConnectorEvent API.
- **DEVICE_EVENT**: Sends the received device events.
- **DEVICE_COMMAND_REQUEST**: Command requests initiated from the device. For example, WebRTC workflows.

The connector can also forward device events using the SendConnectorEvent API, with an optional `userId` parameter.

> **Note**
>
> For GeneralAuthorization: When using GeneralAuthorization, for each Secrets ARN and version, the userId needs to be unique.

- For device events with a `userId`:

  Sample SendConnectorEvent API Request and Response:

```
Request:

{
    "UserId": "*****",
    "Operation": "DEVICE_EVENT",
    "OperationVersion": "1.0",
    "StatusCode": 200,
    "ConnectorId": "****",
    "ConnectorDeviceId": "***",
    "TraceId": "***",
    "MatterEndpoint": {
        "id": "**",
        "clusters": [{
            .....
            }
        }]
    }
```

```
}

Response:

{
    "ConnectorId": "string"
}
```

- For device events without a `userId`:

  Sample SendConnectorEvent API Request and Response:

```
Request:

{
    "Operation": "DEVICE_EVENT",
    "OperationVersion": "1.0",
    "StatusCode": 200,
    "ConnectorId": "*****",
    "ConnectorDeviceId": "****",
    "TraceId": "****",
    "MatterEndpoint": {
        "id": "**",
        "clusters": [{
            ....
        }]
    }
}

Response:

{
    "ConnectorId": "string"
}
```

To remove the link between a particular `managedThing` and an account association, use the deregister mechanism:

Sample DeregisterAccountAssociation API Request and Response:

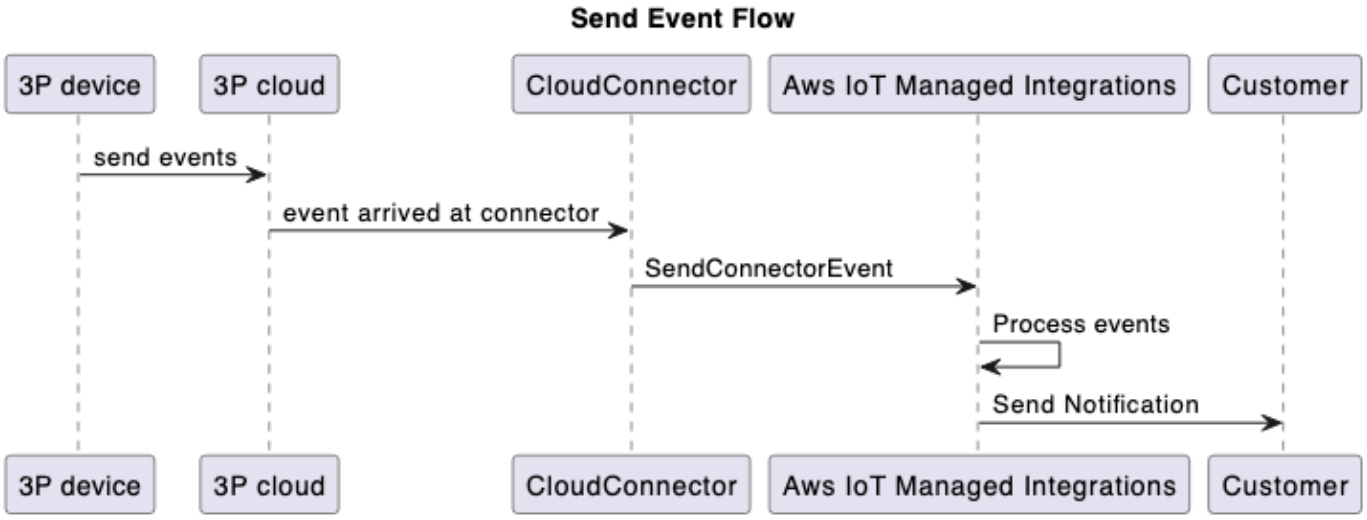
```
Request:
```

```
{
    "AccountAssociationId": "****",
    "ManagedThingId": "****"
}

Response:

HTTP/1.1 200 // Empty body
```

Send event flow:



7. **Update connector status to "Listed" to make it visible to other managed integrations customers**

   By default, connectors are private and only visible to the AWS account that created them. You can choose to make a connector visible to other managed integrations customers.

   To share your connector with other users, use the **Make visible** option in the AWS Management Console on the connector details page to submit your connector ID to AWS for review. Once approved, the connector is available to all managed integrations users in the same AWS Region. Additionally, you can restrict access to specific AWS account IDs by modifying the access policy on the connector's associated AWS Lambda function. To ensure your connector is usable by other customers, manage the IAM access permissions on your Lambda function from other AWS accounts to your visible connector.

Review the AWS service terms and your organization's policies that govern connector sharing and access permissions before making connectors visible to other managed integrations customers.

General/Custom Authorization requirements

General Authorization enables access to devices, allowing you to control multiple devices using credentials without requiring individual user credentials. Unlike OAuth 2.0, which provides user-level authorization, General Authorization uses pre-shared keys, tokens, or other authorization mechanisms stored in AWS Secrets Manager.

With General Authorization, your C2C connector can support authorization mechanisms beyond OAuth 2.0, including API keys, bearer tokens, and custom authorization schemes provided by third-party platforms. This approach is particularly useful for scenarios where devices are managed at a program or organization level rather than by individual end users.

How C2C connectors use secrets for General Authorization

AWS Secrets Manager is a secret storage service that protects sensitive credentials such as API keys and tokens. Secrets are encrypted using AWS Key Management Service keys. For more information, see the AWS Secrets Manager User Guide.

For General Authorization, you store authorization credentials in Secrets Manager, and grant your C2C connector permission to access these secrets. When managed integrations invokes your connector, it provides the Secrets Manager ARN and version ID. Your connector retrieves the secret value and uses it to authenticate with the third-party platform.

This approach ensures that managed integrations never handles your long-term credentials directly. Your connector maintains full control over credential management and token generation, making the solution extensible to any authorization mechanism supported by your third-party platform.

> **Important**
>
> We recommend that you don't log sensitive credentials or tokens in any logs. If however they are stored in logs, we recommend that you use CloudWatch Logs data protection policies to mask the tokens in the logs. For more information, see Help protect sensitive log data with masking.

How to create a secret for General Authorization

To create a secret for General Authorization, follow the steps in Create an AWS Secrets Manager secret in the AWS Secrets Manager User Guide.

You must create your secret with a customer-managed AWS KMS key for your C2C connector to read the secret value. For more information, see Permissions for the AWS KMS key in the AWS Secrets Manager User Guide.

Store your authorization credentials in the secret using a JSON structure. The exact format depends on your third-party platform's authorization requirements. Common examples include:

- API keys: `{"apiKey": "your-api-key-value"}`
- Bearer tokens: `{"token": "your-bearer-token"}`
- Username and password: `{"username": "your-username", "password": "your-password"}`
- Machine-to-machine OAuth credentials: `{"client_id": "your-client-id", "client_secret": "your-client-secret", "audience": "your-audience"}`

You must also configure the IAM policies described in the following section to grant your C2C connector access to retrieve the secret.

Grant access for C2C connector to retrieve the secret

To allow managed integrations to retrieve the secret value from Secrets Manager, include the following permissions in the resource policy for the secret when you create it:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "c2c-connector-account-id"
      },
      "Action": "secretsmanager:GetSecretValue",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
```

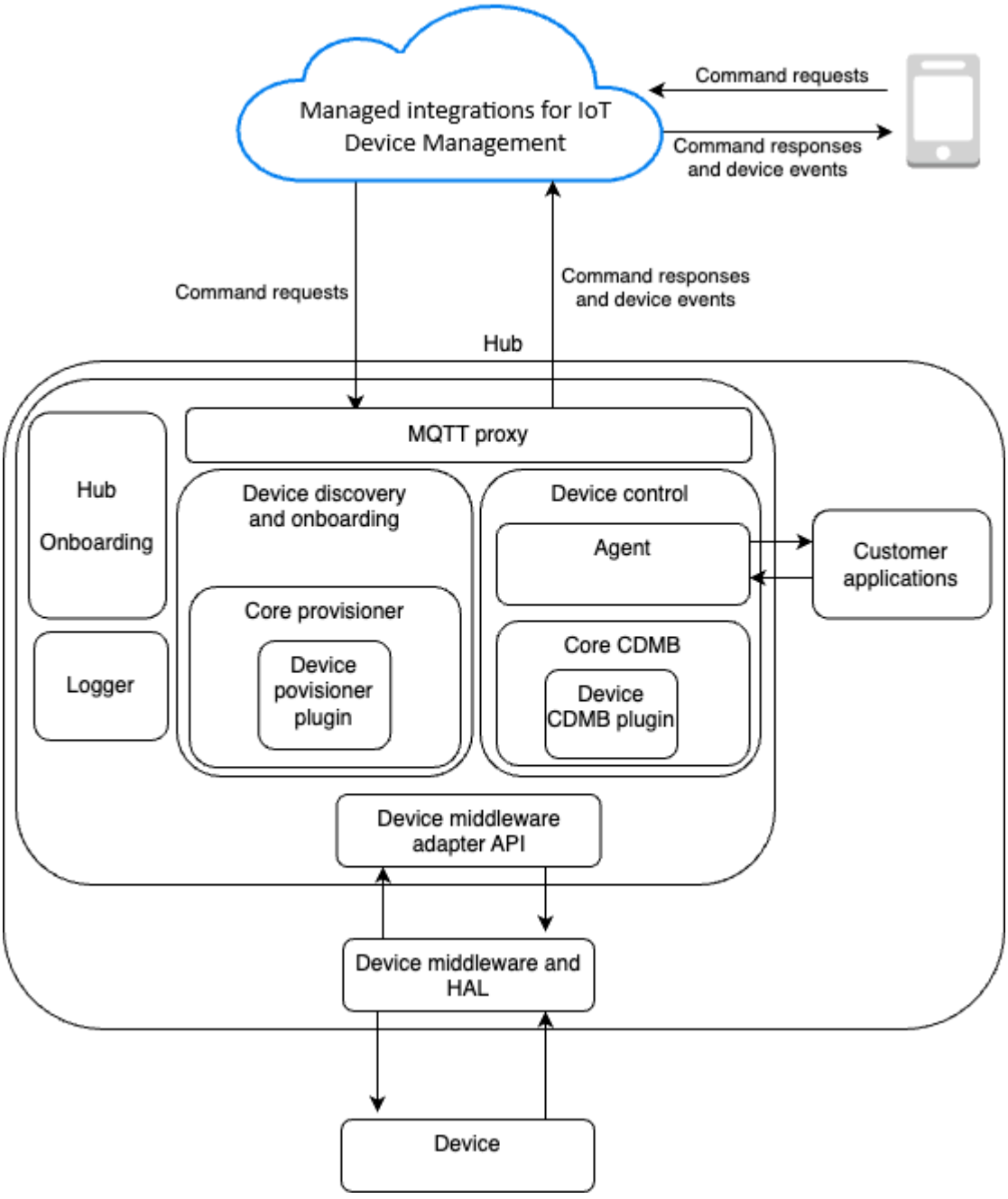
```
          "aws:SourceArn": "arn:aws:iotmanagedintegrations:region:account-id:account-
association/account-association-id"
        }
      }
    }
  ]
}
```

This policy grants managed integrations permission to retrieve the secret value on behalf of your C2C connector. The condition key helps prevent the confused deputy problem by ensuring that only requests originating from your specific account association can access the secret.

Managed Integrations Hub SDK

Use the topics in this section to learn how to onboard and control IoT hub devices using the managed integrations Hub SDK. For more information about the managed integrations End device SDK, see the Managed Integrations End device SDK.

Hub SDK architecture



Device onboarding

Review how the Hub SDK components support device onboarding before you begin working with Managed Integrations. This section covers the essential architectural components you need for device onboarding, including how the core provisioner and protocol-specific plugins work together to handle device authentication, communication, and user setup.

Hub SDK components for device onboarding

SDK components

- Core provisioner
- Protocol-specific provisioner plugins
- Protocol-specific middleware

Core provisioner

The core provisioner is the central component that orchestrates device onboarding in your IoT hub deployment. It coordinates all communication between Managed Integrations and your protocol-specific provisioner plugins, ensuring secure and reliable device onboarding. When you onboard a device, the core provisioner handles the authentication flow, manages MQTT messaging, and processes device requests through these functions:

MQTT connection

Creates connections with the MQTT broker for cloud topic publishing and subscribing.

Message queue and handler

Processes incoming add and remove device requests in sequence.

Protocol plugin interface

Works with protocol-specific provisioner plugins for device onboarding by managing authentication and radio joining modes.

Hub SDK client APIs

Receive and forward device capability reports from protocol-specific CDMB plugins to Managed Integrations.

Protocol-specific provisioner plugins

Protocol-specific provisioner plugins are libraries that manage device onboarding for different communication protocols. Each plugin translates commands from the core provisioner into protocol-specific actions for your IoT devices. These plugins perform:

- Protocol-specific middleware initialization
- Radio joining mode configuration based on core provisioner requests
- Device removal through middleware API calls

Protocol-specific middleware

Protocol-specific middleware acts as a translation layer between your device protocols and Managed Integrations. This component processes communication in both directions—receiving commands from the provisioner plugins and sending them to protocol stacks, while also collecting responses from devices and routing them back through the system.

Device onboarding flows

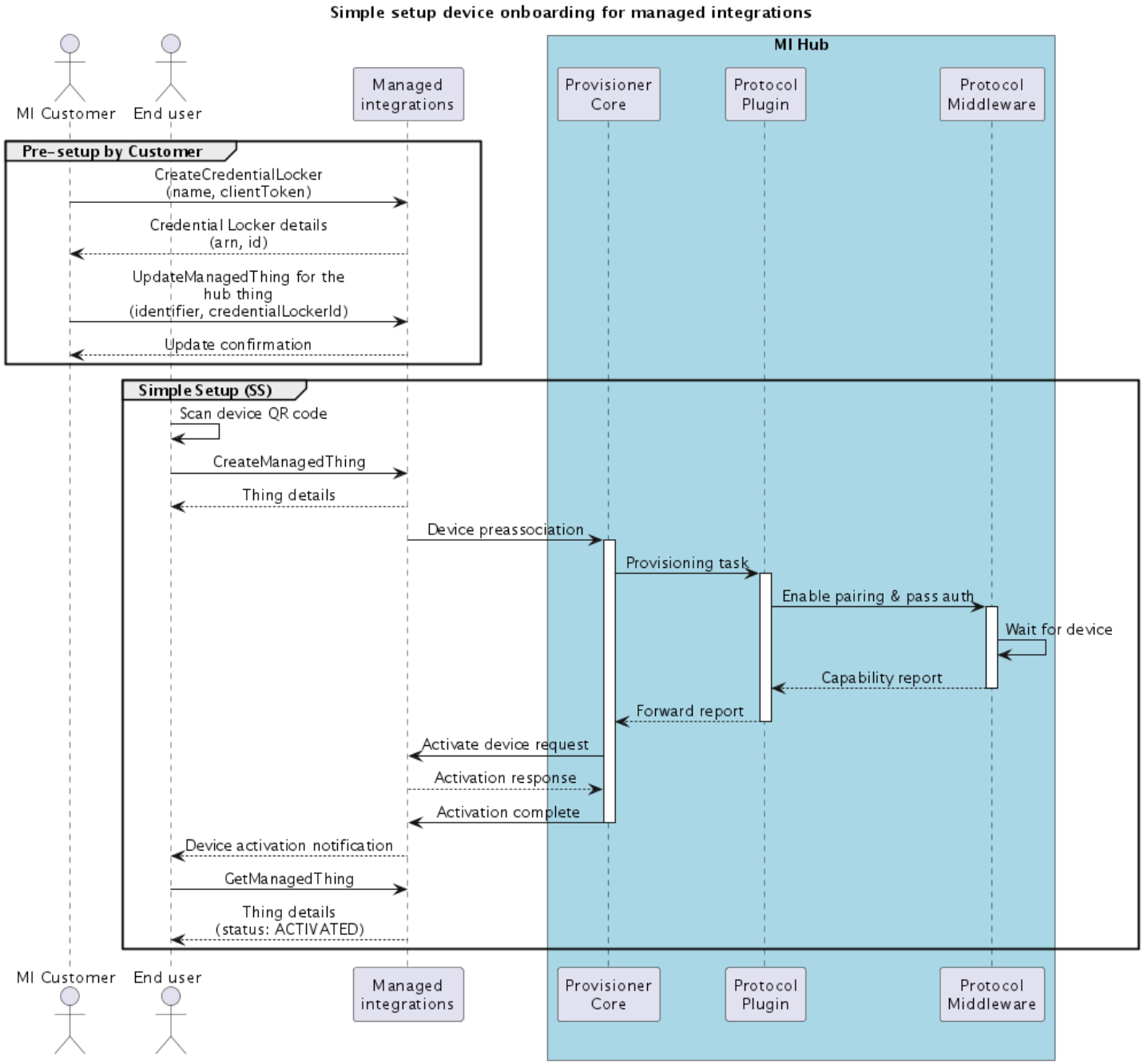
Review the sequence of operations that occur when you onboard devices using the Hub SDK. This section displays how components interact during the onboarding process and outlines the supported onboarding methods.

Onboarding flows

- Simple setup (SS)
- Zero-touch setup (ZTS)
- User guided setup (UGS)
- WiFi Simple Setup (WSS)

Simple setup (SS)

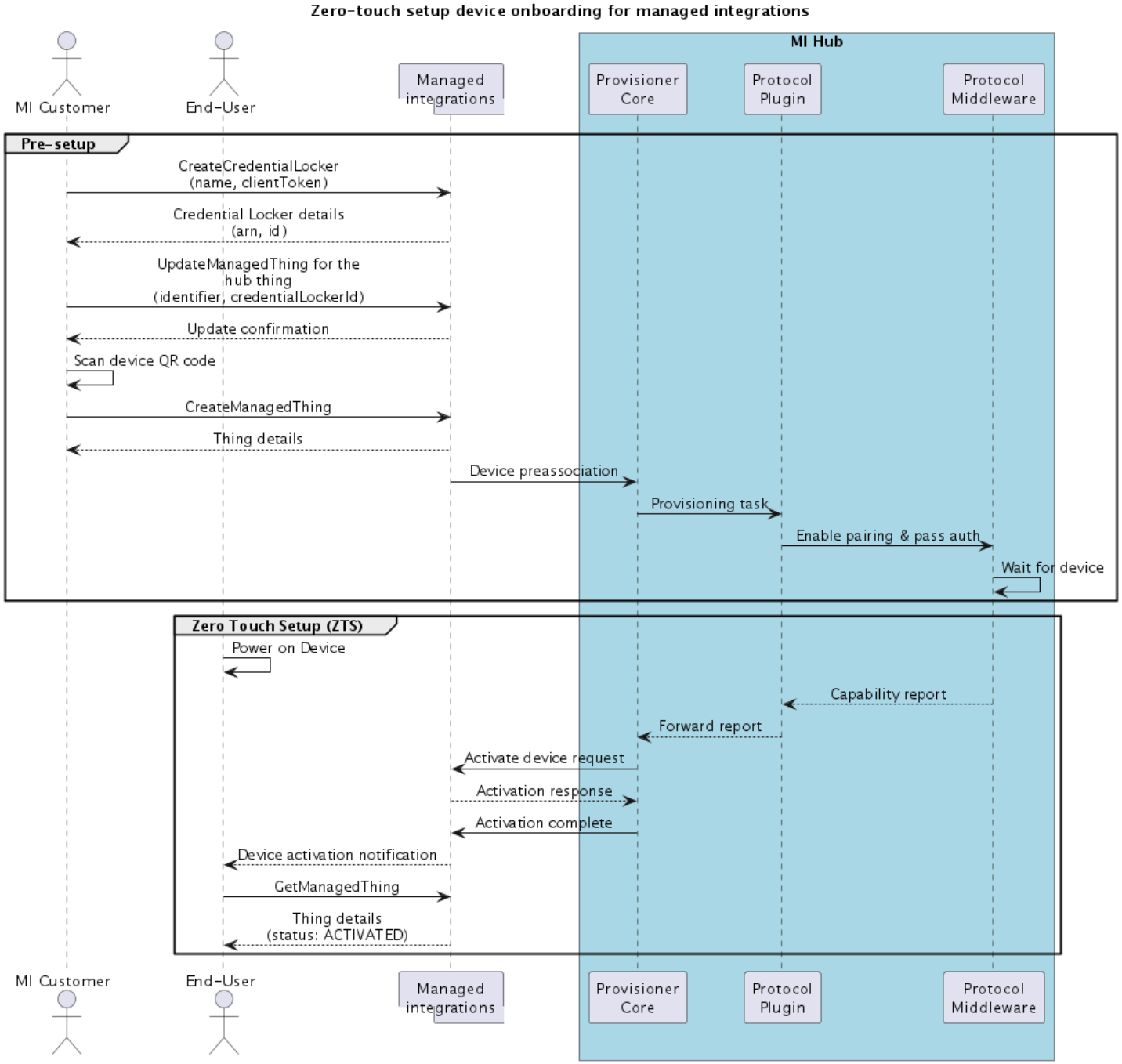
The end user powers on the IoT device and scans its QR code using the device manufacturer application. The device is then enrolled onto the Managed Integrations cloud and connects to the IoT hub.



Zero-touch setup (ZTS)

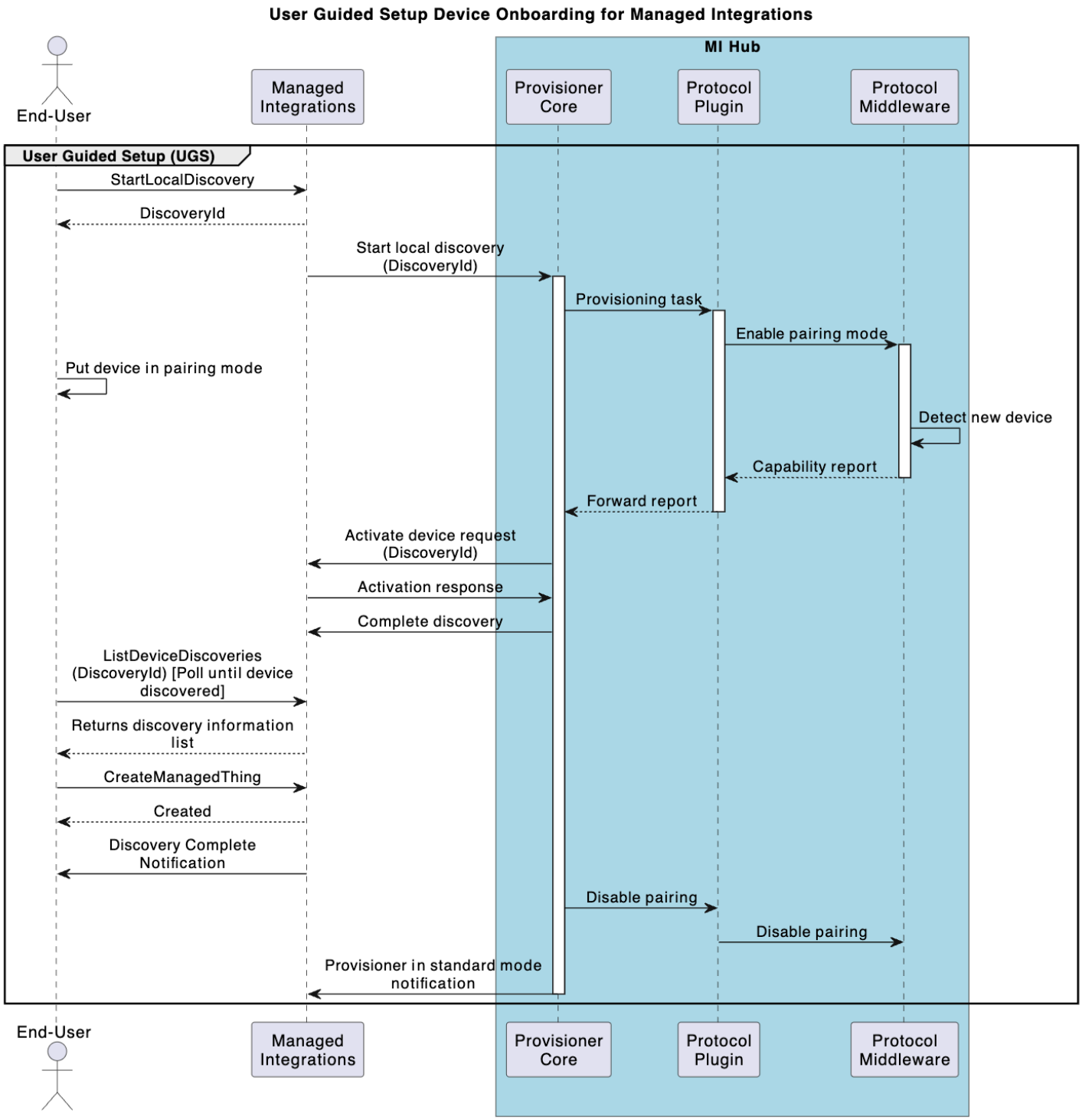
Zero-touch setup (ZTS) streamlines device onboarding by pre-associating the device upstream in the supply chain. For example, instead of end-users scanning the device QR code, this step is completed earlier to pre-link devices to customer accounts. For example, this step can be completed at the fulfillment center.

When the end user receives and powers on the device, it automatically enrolls in the managed integrations cloud and connects to the IoT hub without requiring any additional setup actions.



User guided setup (UGS)

The end user powers on the device and follows interactive steps to onboard it to Managed Integrations. This might include pressing a button on the IoT hub, using a device manufacturer app, or pressing buttons on both the hub and device. You can use this method if Simple setup fails.



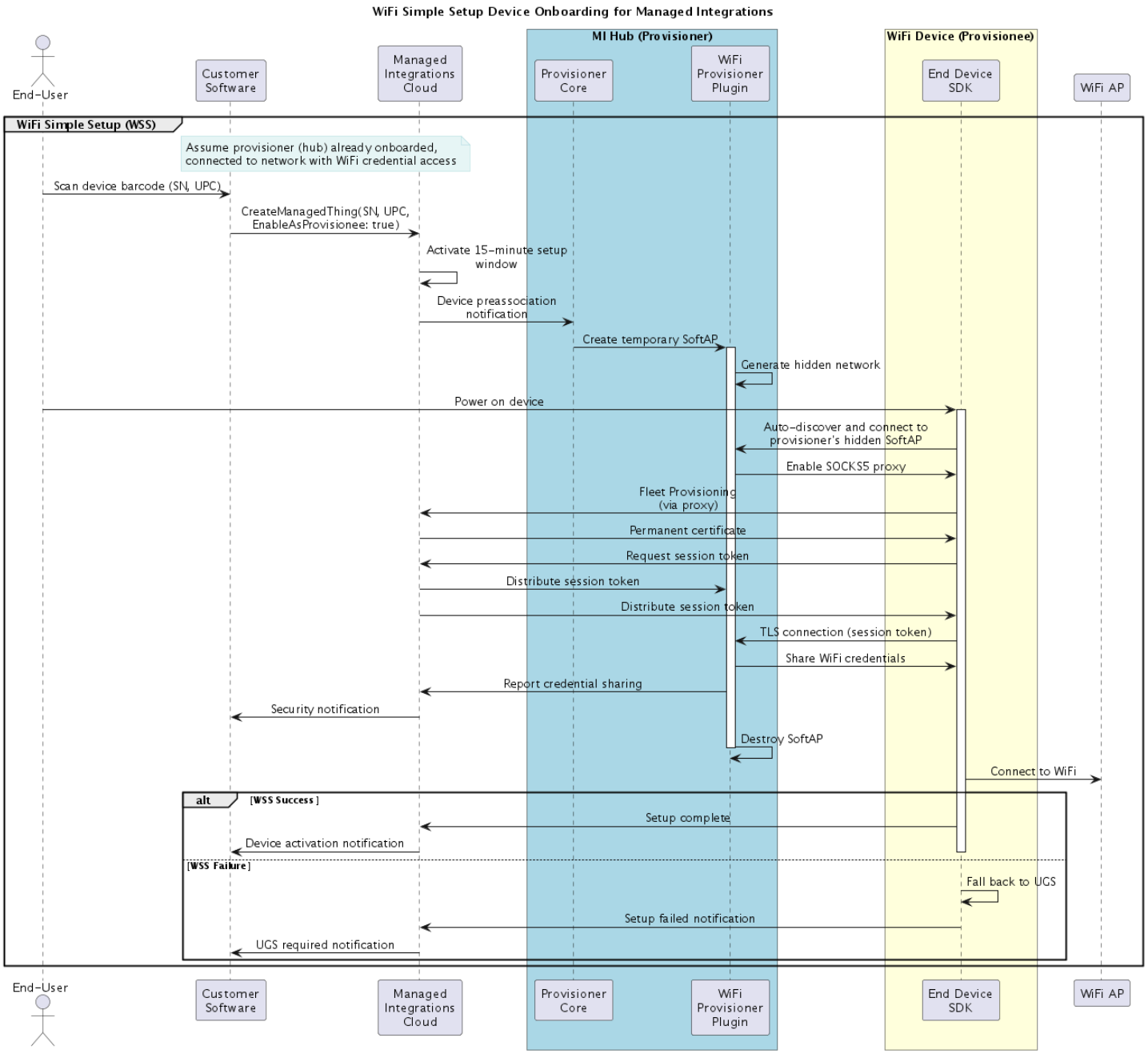
WiFi Simple Setup (WSS)

WiFi Simple Setup (WSS) eliminates the need for manual WiFi configuration by automatically provisioning network credentials. For WiFi-enabled devices, WSS automates WiFi credential provisioning. The end user scans the device barcode (SN and UPC) using the mobile application,

activating a 15-minute setup window. When the device is powered on, it automatically discovers the provisioner's temporary network, completes cloud authentication, and securely receives WiFi credentials from the provisioner. The device then connects to the WiFi network without requiring manual password entry or network selection.

If WSS is unavailable or fails (for example, no provisioner available or connection timeout), the system automatically falls back to User Guided Setup, providing a seamless user experience with manual setup guidance. For more information about WiFi Simple Setup, see WiFi Simple Setup to onboard and operate devices.

The following diagram shows the WiFi Simple Setup flow:



Device control

Managed Integrations handles device registration, command execution, and control. You can build end-user experiences without knowledge of device-specific protocols using its vendor and protocol-agnostic device management.

With device control, you can view and modify device states, such as light bulb brightness or door position. The feature emits events for state changes, which you can use for analytics, rules, and monitoring.

Key features

Modify or read device state

View and change device attributes based on device types. You can access:

- **Device state:** Current device attribute values
- **Connectivity state:** Device reachability status
- **Health status:** System values like battery level and signal strength (RSSI)

State change notification

Receive events when device attributes or connectivity states change, such as light bulb brightness adjustments or door lock status changes.

Offline mode

Devices communicate with other devices on the same IoT hub even without internet connectivity. Device states synchronize with the cloud when connectivity resumes.

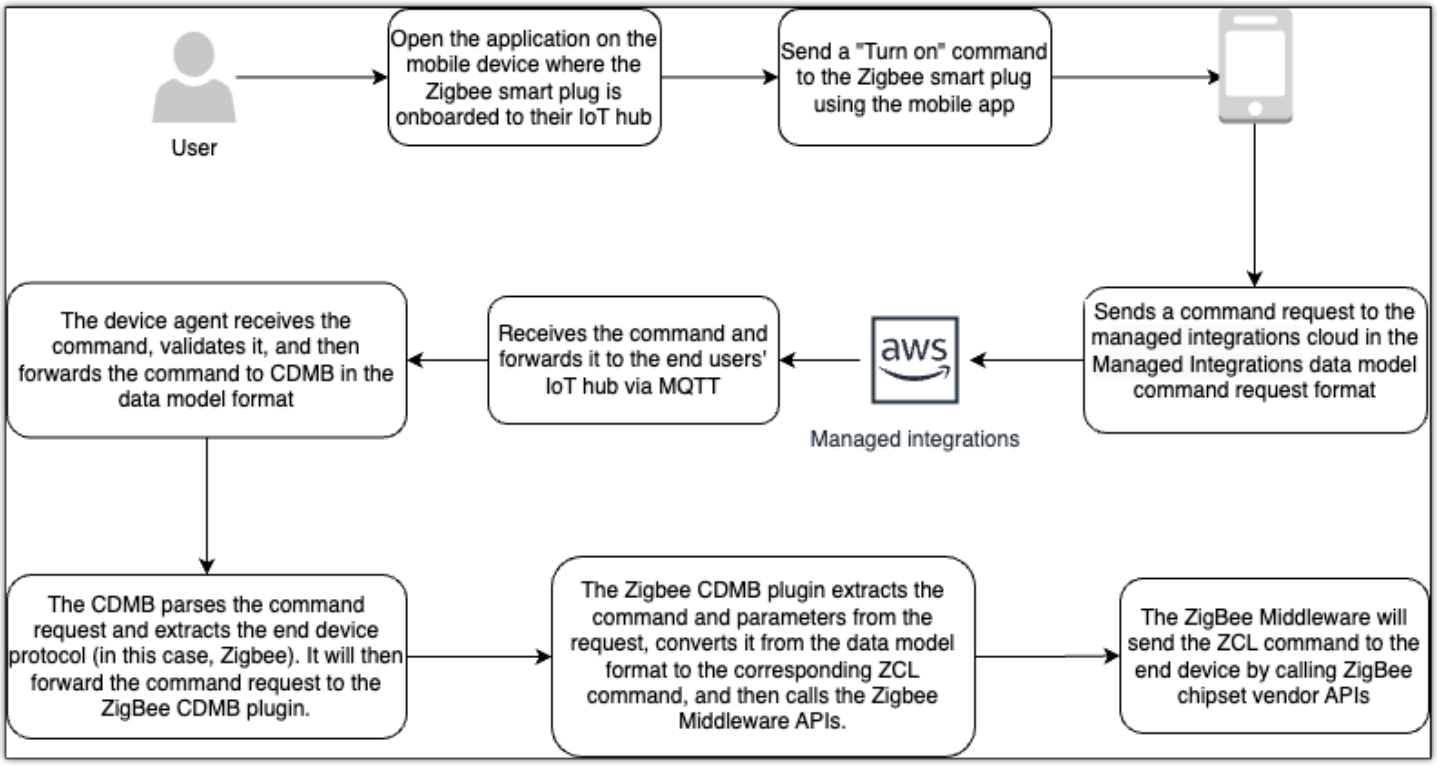
State synchronization

Track state changes from multiple sources, device manufacturer apps, and manual device adjustments.

Review the Hub SDK components and processes you need to control devices through Managed Integrations. This topic describes how the Edge Agent, Common Data Model Bridge (CDMB), and protocol-specific plugins work together to handle device commands, manage device states, and process responses across different protocols.

Device control flows

The following diagram demonstrates the end-to-end device control flow by describing how an end user turns on a Zigbee smart plug.



Hub SDK components for device control

The Hub SDK architecture uses the following components to process and route device control commands in your IoT implementation. Each component plays a specific role in translating cloud commands into device actions, managing device states, and handling responses. The following sections detail how these components work together in your deployment:

The Hub SDK consists of the following components, and facilitates device onboarding and control on IoT hubs.

Primary components:

Edge agent

Acts as a gateway between the IoT hub and Managed Integrations.

Common data model bridge (CDMB)

Translates between the AWS data model and local protocol data models like Z-Wave and Zigbee. It includes a core CDMB and protocol-specific CDMB plugins.

Provisioner

Handles device discovery and onboarding. It includes a core provisioner and protocol-specific provisioner plugins for protocol-specific onboarding tasks.

Secondary components

Hub onboarding

Provisions the hub with client certificates and keys for secure cloud communication.

MQTT proxy

Provides MQTT connections to the Managed Integrations cloud.

Logger

Writes logs locally or to the Managed Integrations cloud.

Install and validate the managed integrations Hub SDK

Choose between the following deployment methods to install the managed integrations Hub SDK on your devices—AWS IoT Greengrass for automated deployment or a manual script installation. This section describes the setup and validation steps for both approaches.

Before you begin, download the Managed Integrations Hub SDK version 2.4.0 for aarch64. The deployment methods in this section install the SDK from this archive. For more information about running the SDK from a prebuilt Raspberry Pi image instead, see Managed Integrations Hub SDK image on Raspberry Pi.

Deployment methods

- Install the Hub SDK with AWS IoT Greengrass
- Deploy the Hub SDK with a script
- Deploy Hub SDK with systemd

Install the Hub SDK with AWS IoT Greengrass

Deploy the managed integrations Hub SDK components for your devices using AWS IoT Greengrass (Java Version).

> **Note**
>
> You must have already set up and have an understanding of AWS IoT Greengrass. For more information, see What is AWS IoT Greengrass in the *AWS IoT Greengrass developer guide documentation*.

The AWS IoT Greengrass user must have permission to modify the following directories:

- `/dev/aipc`
- `/data/aws/iotmi/config`
- `/data/ace/kvstorage`

Topics

- Deploy components locally
- Cloud deployment
- Verify hub provisioning
- Verify CDMB operation
- Verify LPW-Provisioner operation

Deploy components locally

Use the CreateDeployment AWS IoT Greengrass API on your device to deploy the Hub SDK components. The version numbers are not static and can vary based on the version you use at the time. Use the following format for **`version`**: com.amazon.IoTManagedIntegrationsDevice.AceCommon=`0.2.0`.

```
/greengrass/v2/bin/greengrass-cli deployment create \
--recipeDir recipes \
--artifactDir artifacts \
-m "com.amazon.IoTManagedIntegrationsDevice.AceCommon=version" \
-m "com.amazon.IoTManagedIntegrationsDevice.HubOnboarding=version" \
-m "com.amazon.IoTManagedIntegrationsDevice.AceZigbee=version" \
-m "com.amazon.IoTManagedIntegrationsDevice.LPW-Provisioner=version" \
-m "com.amazon.IoTManagedIntegrationsDevice.Agent=version" \
-m "com.amazon.IoTManagedIntegrationsDevice.MQTTProxy=version" \
-m "com.amazon.IoTManagedIntegrationsDevice.CDMB=version" \
```

```
-m "com.amazon.IoTManagedIntegrationsDevice.AceZwave=version"
```

Cloud deployment

Follow the instructions in the AWS IoT Greengrass developer guide to perform the following steps:

1. Upload artifacts to Amazon S3.
2. Update recipes to include the Amazon S3 artifact location.
3. Create a cloud deployment to the device for the new components.

Verify hub provisioning

Confirm successful provisioning by checking your configuration file. Open the `/data/aws/iotmi/config/iotmi_config.json` file and verify the state is set to `PROVISIONED`.

Verify CDMB operation

Check the logs file for CDMB startup messages and successful initialization. The *`logs file`* location can vary depending on where AWS IoT Greengrass is installed.

```
tail -f -n 100 /greengrass/v2/logs/com.amazon.IoTManagedIntegrationsDevice.CDMB.log
```

Example

```
[2024-09-06 02:31:54.413758906][IoTManagedIntegrationsDevice_CDMB][info] Successfully
 subscribed to topic: south/bF|gi_044F8821D0193608C8D5BF80858E20A56E3A8490/control
[2024-09-06 02:31:54.513956059][IoTManagedIntegrationsDevice_CDMB][info] Successfully
 subscribed to topic: south/bF|gi_044F8821D0193608C8D5BF80858E20A56E3A8490/setup
```

Verify LPW-Provisioner operation

Check the logs file for LPW-Provisioner startup messages and successful initialization. The *`logs file`* location can vary depending on where AWS IoT Greengrass is installed.

```
tail -f -n 100 /greengrass/v2/logs/com.amazon.IoTManagedIntegrationsDevice.LPW-
Provisioner.log
```

Example

```
[2024-09-06 02:33:22.068898877][LPWProvisionerCore][info] Successfully subscribed to
 topic: south/bF|gi_044F8821D0193608C8D5BF80858E20A56E3A8490/setup
```

Deploy the Hub SDK with a script

Deploy the managed integrations Hub SDK components manually using installation scripts, then validate the deployment. This section describes the script execution steps and verification process.

Topics

- Prepare your environment
- Run the Hub SDK script
- Verify hub provisioning
- Verify agent operation
- Verify LPW-Provisioner operation

Prepare your environment

Complete these steps before running the SDK installation script:

1. Create a folder named `middleware` inside the `artifacts` folder.
2. Copy your hub middleware files to the `middleware` folder.
3. Run the initialization commands before starting the SDK.

> **Important**
>
> Repeat the initialization commands after each hub reboot.

```
#Get the current user
_user=$(whoami)

#Get the current group
_grp=$(id -gn)

#Display the user and group
echo "Current User: $_user"
```

```
echo "Current Group: $_grp"

sudo mkdir -p /dev/aipc/
sudo chown -R $_user:$_grp /dev/aipc
sudo mkdir -p /data/ace/kvstorage
sudo chown -R $_user:$_grp /data/ace/kvstorage
```

Run the Hub SDK script

Navigate to the artifacts directory and run the `start_iotmi_sdk.sh` script. This script launches the hub SDK components in the correct sequence. Review the following example logs to verify successful startup:

> **Note**
>
> Logs for all the components running can be found inside the `artifacts/logs` folder.

```
hub@hub-293ea release_Oct_17$ ./start_iotmi_sdk.sh
-------Stopping SDK running processes---
DeviceAgent: no process found
------Starting SDK-------
-------Creating logs directory----------
Logs directory created.
-------Verifying Middleware paths-------
All middleware libraries exist
-------Verifying Middleware pre reqs---
AIPC and KVstroage directories exist
-------Starting HubOnboarding-----------
-------Starting MQTT Proxy-----------
-------Starting Event Manager-----------
-------Starting Zigbee Service----------
-------Starting Zwave Service----------
/data/release_Oct_17/middleware/AceZwave/bin /data/release_Oct_17
/data/release_Oct_17
-------Starting CDMB--------------------
-------Starting Agent--------------------
-------Starting Provisioner--------------
-------Checking SDK status---------------
hub         6199  1.7  0.7 1004952 15568 pts/2   Sl+  21:41   0:00 ./iotmi_mqtt_proxy -
C /data/aws/iotmi/config/iotmi_config.json
```

```
Process 'iotmi_mqtt_proxy' is running.
hub         6225  0.0  0.1 301576  2056 pts/2    Sl+  21:41   0:00 ./middleware/
AceCommon/bin/ace_eventmgr
Process 'ace_eventmgr' is running.
hub         6234  104  0.2 238560  5036 pts/2    Sl+  21:41   0:38 ./middleware/
AceZigbee/bin/ace_zigbee_service
Process 'ace_zigbee_service' is running.
hub         6242  0.4  0.7 1569372 14236 pts/2   Sl+  21:41   0:00 ./zwave_svc
Process 'zwave_svc' is running.
hub         6275  0.0  0.2 1212744 5380 pts/2    Sl+  21:41   0:00 ./DeviceCdmb
Process 'DeviceCdmb' is running.
hub         6308  0.6  0.9 1076108 18204 pts/2   Sl+  21:41   0:00 ./
IoTManagedIntegrationsDeviceAgent
Process 'DeviceAgent' is running.
hub         6343  0.7  0.7 1388132 13812 pts/2   Sl+  21:42   0:00 ./
iotmi_lpw_provisioner
Process 'iotmi_lpw_provisioner' is running.
------Successfully Started SDK----
```

Verify hub provisioning

Check that the `iot_provisioning_state` field in `/data/aws/iotmi/config/iotmi_config.json` is set to `PROVISIONED`. .

Verify agent operation

Check the logs file for agent startup messages and successful initialization.

```
tail -f -n 100 logs/agent_logs.txt
```

Example

```
[2024-09-06 02:31:54.413758906][Device_Agent][info] Successfully subscribed to topic:
 south/bF|gi_044F8821D0193608C8D5BF80858E20A56E3A8490/control
[2024-09-06 02:31:54.513956059][Device_Agent][info] Successfully subscribed to topic:
 south/bF|gi_044F8821D0193608C8D5BF80858E20A56E3A8490/setup
```

> **Note**
>
> Check that the `iotmi.db` database exists in your `artifacts` directory.

Verify LPW-Provisioner operation

Check the logs file for `LPW-Provisioner` startup messages and successful initialization.

```
tail -f -n 100 logs/provisioner_logs.txt
```

The following code shows an example.

```
[2024-09-06 02:33:22.068898877][LPWProvisionerCore][info] Successfully subscribed to
 topic: south/bF|gi_044F8821D0193608C8D5BF80858E20A56E3A8490/setup
```

Deploy Hub SDK with systemd

> **Important**
>
> Follow the `readme.md` in the `hubSystemdSetup` directory of the release.tgz file for the latest updates.

This section describes the scripts and processes for deploying and configuring services on a Linux-based hub device.

Overview

The deployment process consists of two main scripts:

- `copy_to_hub.sh`: Runs on the host machine to copy necessary files to the hub
- `setup_hub.sh`: Runs on the hub to configure the environment and deploy services

Additionally, `systemd/deploy_iotshd_services_on_hub.sh` handles process bootstrap sequence and process permission management, and is automatically triggered by `setup_hub.sh`.

Prerequisites

The listed prerequistes are required for successful deployment.

- systemd service is available on the hub

- SSH access to the hub device
- Sudo privileges on the hub device
- `scp` utility installed on host machine
- `sed` utility installed on host machine
- unzip utility installed on host machine

File structure

The file structure is designed to facilitate the organization and management of its various components, enabling efficient access and navigation of the content.

```
hubSystemdSetup/
### README.md
### copy_to_hub.sh
### setup_hub.sh
### iotshd_config.json  # Sample configuration file
### local_certs/  # Directory for DHA certificates
### systemd/
    ### *.service.template  # Systemd service templates
    ### deploy_iotshd_services_on_hub.sh
```

In the SDK release tgz file, the overall file structure is:

```
IoT-managed-integrations-Hub-SDK-aarch64-v1.0.0.tgz
###package/
    ###greengrass/
        ###artifacts/
        ###recipes/
    ###hubSystemdSetup/
        ### REAME.md
        ### copy_to_hub.sh
        ### setup_hub.sh
        ### iotshd_config.json  # Sample configuration file
        ### local_certs/  # Directory for DHA certificates
        ### systemd/
            ### *.service.template  # Systemd service templates
            ### deploy_iotshd_services_on_hub.sh
```

Initial setup

Extract the SDK package

```
tar -xzf managed-integrations-Hub-SDK-vVersion-linux-aarch64-timestamp.tgz
```

Navigate to the extracted directory and prepare the package:

```
# Create package.zip containing required artifacts
zip -r package.zip package/greengrass/artifacts
# Move package.zip to the hubSystemdSetup directory
mv package.zip ../hubSystemdSetup/
```

Add device configuration files

Follow the two steps listed to create the device configuration files, and copy them to the hub.

1. Add device configuration files to create the device configuration files needed. The SDK uses this file for its function.
2. Copy the configuration files to copy the created configuration files to the hub.

Copy files to the hub

Run the deployment script from your host machine:

```
chmod +x copy_to_hub.sh
./copy_to_hub.sh hub_ip_address package_file
```

Example

```
./copy_to_hub.sh 192.168.50.223 ~/Downloads/EAR3-package.zip
```

This copies:

- The package file (renamed to package.zip on the hub)
- Configuration files
- Certificates
- Systemd service files

Set up hub

After the files are copied, SSH into the hub and run the setup script:

```
ssh hub@hub_ip
chmod +x setup_hub.sh
sudo ./setup_hub.sh
```

User and group configurations

By default, we use the user **hub** and group **hub** for the SDK components. There are multiple ways to configure them:

- Use a custom user/group:

  ```
  sudo ./setup_hub.sh --user=USERNAME --group=GROUPNAME
  ```

- Create them manually before running the setup script:

  ```
  sudo groupadd -f GROUPNAME
  sudo useradd -r -g GROUPNAME USERNAME
  ```

- Add the commands in `setup_hub.sh`.

Manage services

To restart all the services, run the following script from hub:

```
sudo /usr/local/bin/deploy_iotshd_services_on_hub.sh
```

The setup script will create necessary directories, set appropriate permissions, and deploy services automatically. If you're not using SSH/SCP, you must modify `copy_to_hub.sh` for your specific deployment method. Ensure all certificate files and configurations are properly set up before deployment.

Onboard your hubs to Managed Integrations

Set up your hub devices to communicate with Managed Integrations by configuring the required directory structure, certificates, and device configuration files. This section describes how the hub onboarding subsystem components work together, where to store certificates and configuration

files, how to create and modify the device configuration file, and the steps to complete the hub provisioning process.

Hub onboarding subsystem

The hub onboarding subsystem uses these core components to manage device provisioning and configuration:

Hub onboarding component

Manages the hub onboarding process by coordinating hub state, provisioning approach, and authentication materials.

Device config file

Stores essential hub configuration data on the device, including:

- Device provisioning state (provisioned or non-provisioned)
- Certificate and key locations
- Authentication information Other SDK processes, such as the MQTT proxy, reference this file to determine hub state and connection settings.

Certificate handler interface

Provides a utility interface for reading and writing device certificates and keys. You can implement this interface to work with:

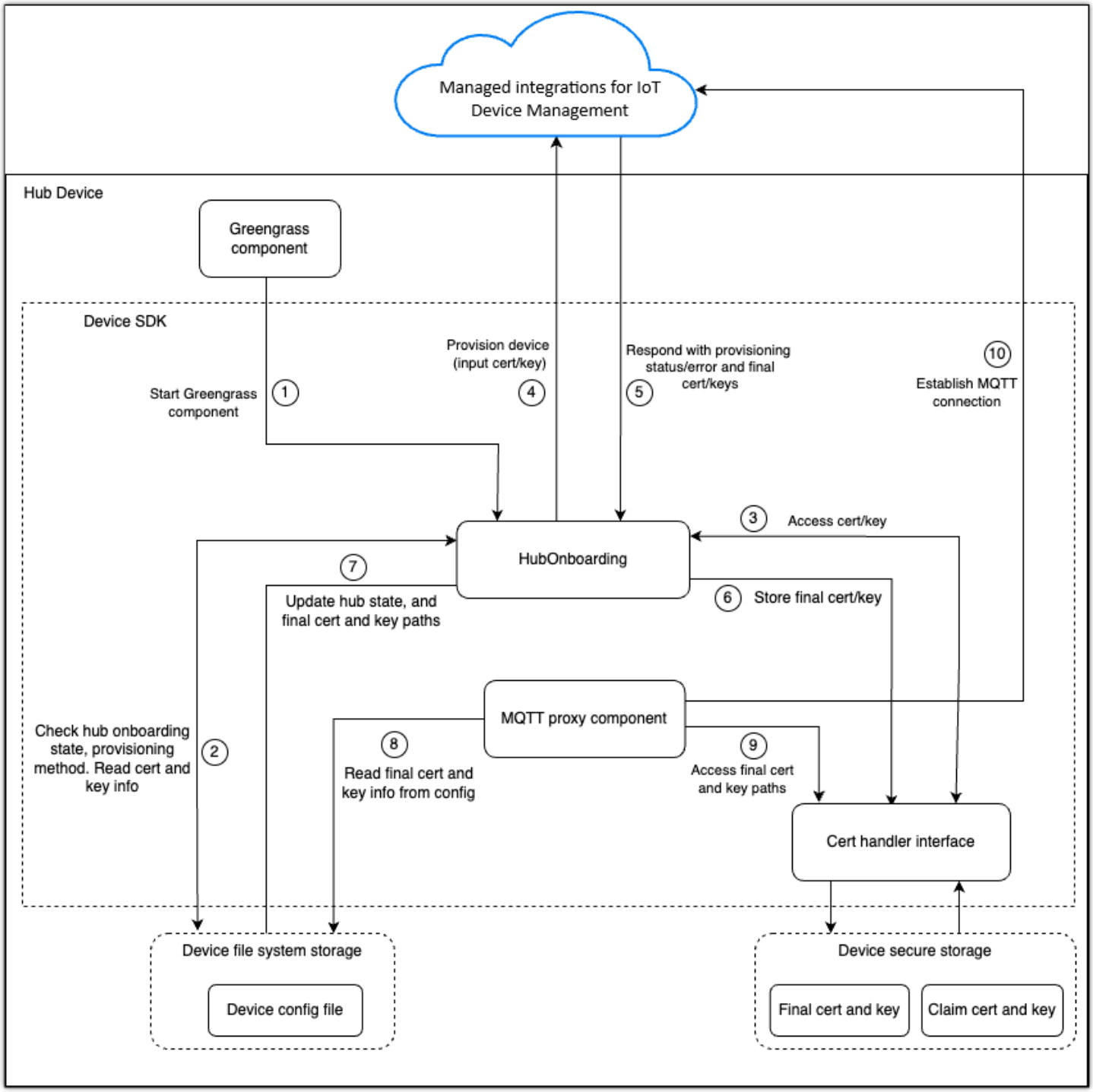
- File system storage
- Hardware security modules (HSM)
- Trusted platform modules (TPM)
- Custom secure storage solutions

MQTT proxy component

Manages device-to-cloud communication using:

- Provisioned client certificates and keys
- Device state information from the config file
- MQTT connections to Managed Integrations

The following diagram describes the hub onboarding subsystem architecture and its components. If you're not using AWS IoT Greengrass, you can disregard that component of the diagram.



Hub onboarding setup

Complete these setup steps for each hub device before you begin the fleet provisioning onboarding process. This section describes how to create managed things, set up directory structures, and configure the required certificates.

Setup steps

- Step 1: Register a custom endpoint
- Step 2: Create a provisioning profile
- Step 3: Create a managed thing (fleet provisioning)
- Step 4: Create the directory structure
- Step 5: Add authentication materials to hub device
- Step 6: Create the device configuration file
- Step 7: Copy the configuration file to your hub

Step 1: Register a custom endpoint

Create a dedicated communication endpoint that your devices use to exchange data with Managed Integrations. This endpoint establishes a secure connection point for all device-to-cloud messaging, including device commands, status updates, and notifications.

To register an endpoint

- Use the RegisterCustomEndpoint API to create an endpoint for device-to-Managed Integrations communication.

  RegisterCustomEndpoint Request example

  ```
  aws iot-managed-integrations register-custom-endpoint
  ```

  Response:

  ```
  {
    [ACCOUNT-PREFIX]-ats.iot.AWS-REGION.amazonaws.com
  }
  ```

  > **Note**
  >
  > Store the endpoint address. You'll need it for future device communication.

  To return the endpoint information, use the `GetCustomEndpoint` API.

For more information, see the RegisterCustomEndpoint API and the GetCustomEndpoint API in the Managed Integrations *API Reference Guide*.

Step 2: Create a provisioning profile

A provisioning profile contains the security credentials and configuration settings your devices need to connect to Managed Integrations.

To create a fleet provisioning profile

- Call the CreateProvisioningProfile API to generate the following:
  - A provisioning template that defines device connection settings
  - A claim certificate and private key for device authentication

> **Important**
>
> Store the claim certificate, private key, and template ID securely. You'll need these credentials to onboard devices to Managed Integrations. If you lose these credentials, you must create a new provisioning profile.

`CreateProvisioningProfile` example request

```
aws iot-managed-integrations create-provisioning-profile \
    --provisioning-type FLEET_PROVISIONING \
    --name PROFILE_NAME
```

Response:

```
{
"Arn":"arn:aws:iotmanagedintegrations:AWS-REGION:ACCOUNT-ID:provisioning-
profile/PROFILE-ID",
    "ClaimCertificate":
  "-----BEGIN CERTIFICATE-----
  MIICiTCCAfICCQD6m7.....w3rrszlaEXAMPLE=
  -----END CERTIFICATE-----",
    "ClaimCertificatePrivateKey":
```

```
  "-----BEGIN RSA PRIVATE KEY-----
  MIICiTCCAfICCQ...3rrszlaEXAMPLE=
 -----END RSA PRIVATE KEY-----",
    "Id": "PROFILE-ID",
    "PROFILE-NAME",
         "ProvisioningType": "FLEET_PROVISIONING"
}
```

Step 3: Create a managed thing (fleet provisioning)

Use the `CreateManagedThing` API to create a managed thing for your hub device. Each hub requires its own managed thing with unique authentication materials. For more information, see the CreateManagedThing API in the Managed Integrations *API Reference*.

When you create a managed thing, specify these parameters:

- `Role`: Set this value to `CONTROLLER` for hubs that do not support command and control, otherwise set to `DEVICE`.
- `AuthenticationMaterialType`: Set this value to `WIFI_SETUP_QR_BAR_CODE`.
- `AuthenticationMaterial`: Include the following fields. You can use either UPC or EAN but not both.
  - SN: The unique serial number for this device
  - UPC: The universal product code for this device
  - EAN: The international article number for this device

> **Important**
>
> Each device must have a unique serial number (SN) in its authentication material.

`CreateManagedThing` Request example:

```
{
 "Role": "CONTROLLER",
 "AuthenticationMaterialType": "WIFI_SETUP_QR_BAR_CODE",
 "AuthenticationMaterial": "SN:123456789524;UPC:829576019524"
}
```

For more information, see CreateManagedThing in the Managed Integrations *API Reference*.

(Optional) Get managed thing

The `ProvisioningStatus` of your managed thing must be `PRE_ASSOCIATED` before you can proceed. For more information on ProvisioningStatus, see Device Provisioning. Use the `GetManagedThing` API to verify that your managed thing exists and is ready for provisioning. For more information, see GetManagedThing in the Managed Integrations *API Reference*.

Step 4: Create the directory structure

Create directories for your configuration files and certificates. By default, the hub onboarding process uses the `/data/aws/iotmi/config/iotmi_config.json`.

You can specify custom paths for certificates and private keys in the configuration file. This guide uses the default path `/data/aws/iotmi/certs`.

```
mkdir -p /data/aws/iotmi/config
mkdir -p /data/aws/iotmi/certs

/data/
    aws/
        iotmi/
            config/
            certs/
```

Step 5: Add authentication materials to hub device

Copy certificates and keys to your hub device, then create a device-specific configuration file. These files establish secure communication between your hub and Managed Integrations during the provisioning process.

To copy claim certificate and key

- Copy these authentication files from your `CreateProvisioningProfile` API response to your hub device:
  - `claim_cert.pem`: The claim certificate (common to all devices)
  - `claim_pk.key`: The private key for the claim certificate

  Place both files in the `/data/aws/iotmi/certs` directory.

> **Important**
>
> When storing certificates and private keys in PEM format, ensure proper formatting by handling newline characters correctly. For PEM-encoded files, the newline characters (`\n`) must be replaced with actual line separators, as simply storing escaped newlines will not be correctly retrieved later.

> **Note**
>
> If you use secure storage, store these credentials in your secure storage location instead of the file system. For more information, see Create a custom certificate handler for secure storage.

Step 6: Create the device configuration file

Create a configuration file that contains unique device identifiers, certificate locations, and provisioning settings. The SDK uses this file during hub onboarding to authenticate your device, manage provisioning status, and store connection settings.

> **Note**
>
> Each hub device requires its own configuration file with unique device-specific values.

Use the following procedure to create or modify your configuration file, and copy it to the hub.

- **Create or modify the configuration file (fleet provisioning)**.

  Configure these required fields in the device configuration file:

  - Certificate paths
    1. `iot_claim_cert_path`: Location of your claim certificate (`claim_cert.pem`)
    2. `iot_claim_pk_path`: Location of your private key (`claim_pk.key`)
    3. Use `SECURE_STORAGE` for both fields when implementing the Secure Storage Cert Handler

- Connection settings
  1. `fp_template_name`: The `ProvisioningProfile` name from earlier.
  2. `endpoint_url`: Your Managed Integrations endpoint URL from the `RegisterCustomEndpoint` API response (same for all devices in a Region).
- Device identifiers
  1. SN: Device serial number that matches your `CreateManagedThing` API call (unique per device)
  2. UPCUniversal product code from your `CreateManagedThing` API call (same for all devices of this product)

```
{
    "ro": {
        "iot_provisioning_method": "FLEET_PROVISIONING",
        "iot_claim_cert_path": "<SPECIFY_THIS_FIELD>",
        "iot_claim_pk_path": "<SPECIFY_THIS_FIELD>",
        "fp_template_name": "<SPECIFY_THIS_FIELD>",
        "endpoint_url": "<SPECIFY_THIS_FIELD>",
        "SN": "<SPECIFY_THIS_FIELD>",
        "UPC": "<SPECIFY_THIS_FIELD>"
    },
    "rw": {
        "iot_provisioning_state": "NOT_PROVISIONED"
    }
}
```

Contents of the configuration file

Review the contents of the `iotmi_config.json` file.

Contents

Key	Values	Added by customer?	Notes
`iot_provisioning_method`	`FLEET_PROVISIONING`	Yes	Specify the provisioning method that you want to use.

Key	Values	Added by customer?	Notes
iot_claim_cert_path	The file path that you specify or SECURE_STORAGE . For example, /data/aws/iotmi/certs/claim_cert.pem	Yes	Specify the file path that you want to use or SECURE_STORAGE .
iot_claim_pk_path	The file path that you specify or SECURE_STORAGE . For example, /data/aws/iotmi/certs/claim_pk.pem	Yes	Specify the file path that you want to use or SECURE_STORAGE .
fp_template_name	The fleet provisioning template name should be equal to the name of the ProvisioningProfile that was used earlier.	Yes	Equal to the name of the ProvisioningProfile that was used earlier
endpoint_url	The endpoint URL for Managed Integrations.	Yes	Your devices use this URL to connect to the Managed Integrations cloud. To obtain this information, use the RegisterCustomEndpoint API.
SN	The device serial number. For example, AIDACKCEVSQ6C2EXAMPLE .	Yes	You must provide this unique information for each device.
UPC	Device universal product code. For example, 841667145075 .	Yes	You must provide this information for the device.

Key	Values	Added by customer?	Notes
`managed_thing_id`	The ID of the managed thing.	No	This information is added later by the onboarding process after hub provisioning.
`iot_provisioning_state`	The provisioning state.	Yes	The provisioning state must be set as `NOT_PROVISIONED` .
`iot_permanent_cert_path`	The IoT certificate path. For example, `/data/aws/iotmi/iot_cert.pem` .	No	This information is added later by the onboarding process after hub provisioning.
`iot_permanent_pk_path`	The IoT private key file path. For example, `/data/aws/iotmi/iot_pk.pem` .	No	This information is added later by the onboarding process after hub provisioning.
`client_id`	The client ID that will be used for MQTT connections.	No	This information is added later by the onboarding process after hub provisioning, for other components to consume.
`mqtt_keep_alive_interval`	Range is 30-1200, and units are in seconds. The default value is 300.	Yes	Use this to set a keep-alive interval for MQTT connections.
`event_manager_upper_bound`	The default value is 500.	No	This information is added later by the onboarding process after hub provisioning, for other components to consume.

Step 7: Copy the configuration file to your hub

Copy your configuration file to `/data/aws/iotmi/config` or your custom directory path. You'll provide this path to the `HubOnboarding` binary during the onboarding process.

For fleet provisioning

```
/data/
    aws/
        iotmi/
            config/
                iotmi_config.json
            certs/
                claim_cert.pem
                claim_pk.key
```

Onboard devices and operate them in hub

Set up your devices to be onboarded to your Managed Integrations hub by creating a managed thing and connecting it to your hub. Devices can be onboarded to a hub through simple setup, user guided setup, or WiFi Simple Setup.

Topics

- Simple setup to onboard and operate devices
- User guided setup to onboard and operate devices
- Capability rediscovery
- WiFi Simple Setup to onboard and operate devices

Simple setup to onboard and operate devices

Set up your devices to be onboarded to your Managed Integrations hub by creating a managed thing and connecting it to your hub. This section describes the steps to complete the device onboarding process using simple setup.

Prerequisites

Complete these steps before attempting to onboard a device:

- Onboard a hub device to the Managed Integrations hub.

- Install the latest version of AWS CLI from the Managed Integrations AWS CLI Command Reference
- Subscribe to DEVICE_LIFE_CYCLE event notifications.

Setup steps

- Step 1: Create a credential locker
- Step 2: Add the credential locker to your hub
- Step 3: Create a managed thing with credentials.
- Step 4: Plug in the device and check its status.
- Step 5: Get Device Capabilities
- Step 6: Send a command to the managed thing
- Step 7: Remove the managed thing from your hub

Step 1: Create a credential locker

Create a credential locker for your device.

To create a credential locker

- Use the `create-credential-locker` command. Executing this command will trigger the creation of all manufacturing resources including the Wi-Fi setup key pair and device certificate.

  create-credential-locker example

  ```
  aws iot-managed-integrations create-credential-locker \
    --name "DEVICE_NAME"
  ```

  Response:

  ```
  {
    "Id": "LOCKER_ID"
    "Arn": "arn:aws:iotmanagedintegrations:AWS_REGION:AWS_ACCOUNT_ID:credential-
  locker/LOCKER_ID
    "CreatedAt": "2025-06-09T13:58:52.977000+08:00"
  }
  ```

For more information, see the create-credential-locker command in the Managed Integrations AWS CLI *Command Reference*.

Step 2: Add the credential locker to your hub

Add the credential locker to your hub.

To add a credential locker to your hub

- Use the following command to add a credential locker to your hub.

```
aws iotmi --region AWS_REGION --endpoint AWS_ENDPOINT update-managed-thing \
--identifier "HUB_MANAGED_THING_ID" --credential-locker-id "LOCKER_ID"
```

Step 3: Create a managed thing with credentials.

Create a managed thing with credentials for your device. Each device requires its own managed thing.

To create a managed thing

- Use the `create-managed-thing` command to create a managed thing for your device.

  create-managed-thing example

```
#ZWAVE:
aws iot-managed-integrations create-managed-thing --role DEVICE \
--authentication-material '900137947003133...' \ #auth material from zwave qr code
--authentication-material-type ZWAVE_QR_BAR_CODE \
--credential-locker-id ${locker_id}

#ZIGBEE:
aws iot-managed-integrations create-managed-thing --role DEVICE \
--authentication-material 'Z:286...$I:A4DC00.' \ #auth material from zigbee qr code
--authentication-material-type ZIGBEE_QR_BAR_CODE \
--credential-locker-id ${locker_id}
```

> **Note**
>
> There are separate commands for Z-wave and Zigbee devices.

Response:

```
{
  "Id": "DEVICE_MANAGED_THING_ID"
  "Arn": "arn:aws:iotmanagedintegrations:AWS_REGION:AWS_ACCOUNT_ID:managed-
thing/DEVICE_MANAGED_THING_ID"
  "CreatedAt": "2025-06-09T13:58:52.977000+08:00"
}
```

For more information, see the create-managed-thing command in the Managed Integrations AWS CLI *Command Reference*.

Step 4: Plug in the device and check its status.

Plug in the device and check its status.

- Use the `get-managed-thing` command to check your device's status. The ProvisioningStatus of your managed thing must be ACTIVATED. For more information on ProvisioningStatus, see Device Provisioning.

get-managed-thing example

```
#KINESIS NOTIFICATION:
{
    "version": "1.0.0",
    "messageId": "4ac684bb7f4c41adbb2eecc1e7991xxx",
    "messageType": "DEVICE_LIFE_CYCLE",
    "source": "aws.iotmanagedintegrations",
    "customerAccountId": "12345678901",
    "timestamp": "2025-06-10T05:30:59.852659650Z",
    "region": "us-east-1",
    "resources": ["XXX"],
    "payload": {
        "deviceDetails": {
```

```
            "id": "1e84f61fa79a41219534b6fd57052XXX",
            "arn": "XXX",
            "createdAt": "2025-06-09T06:24:34.336120179Z",
            "updatedAt": "2025-06-10T05:30:59.784157019Z"
        },
        "status": "ACTIVATED"
    }
}
aws iot-managed-integrations get-managed-thing \
--identifier :"DEVICE_MANAGED_THING_ID"
```

Response:

```
{
  "Id": "DEVICE_MANAGED_THING_ID"
  "Arn": "arn:aws:iotmanagedintegrations:AWS_REGION:AWS_ACCOUNT_ID:managed-
thing/MANAGED_THING_ID"
  "CreatedAt": "2025-06-09T13:58:52.977000+08:00"
}
```

For more information, see the get-managed-thing command in the Managed Integrations AWS CLI *Command Reference*.

Step 5: Get Device Capabilities

Use the `get-managed-thing-capabilities` command to obtain your endpoint ID and view a list of possible actions for your device.

To get a device's capabilities

- Use the `get-managed-thing-capabilities` command and note the endpoint ID.

  get-managed-thing-capabilties example

  ```
  aws iotmi get-managed-thing-capabilities \
  --identifier "DEVICE_MANAGED_THING_ID"
  ```

  Response:

  ```
  {
  ```

```
    "ManagedThingId": "1e84f61fa79a41219534b6fd57052cbc",
    "CapabilityReport": {
        "version": "1.0.0",
        "nodeId": "zw.FCB10009+06",
        "endpoints": [
            {
                "id": "ENDPOINT_ID"
                "deviceTypes": [
                    "On/Off Switch"
                ],
                "capabilities": [
                    {
                        "id": "matter.OnOff@1.4",
                        "name": "On/Off",
                        "version": "6",
                        "properties": [
                            "OnOff"
                        ],
                        "actions": [
                            "Off",
                            "On"
                        ],
                        "events": []
                    }
                    ...
}
```

For more information, see the get-managed-thing-capabilities command in the Managed Integrations AWS CLI *Command Reference*.

Step 6: Send a command to the managed thing

Use the `send-managed-thing-command` command to send a toggle action command to your managed thing.

To send a command to your managed thing

- Use the `send-managed-thing-command` command to send a command to your managed thing.

  send-managed-thing-command example

```
json=$(jq -cr '.|@json') <<EOF
[
  {
    "endpointId": "1",
    "capabilities": [
      {
        "id": "matter.OnOff@1.4",
        "name": "On/Off",
        "version": "1",
        "actions": [
          {
            "name": "Toggle",
            "parameters": {}
          }
        ]
      }
    ]
  }
]
EOF
aws iot-managed-integrations send-managed-thing-command \
--managed-thing-id "DEVICE_MANAGED_THING_ID" --endpoints "ENDPOINT_ID"
```

Note

This example uses jq cli to but you can also pass the entire `endpointId` string

Response:

```
{
"TraceId": "TRACE_ID"
}
```

For more information, see the send-managed-thing-command command in the Managed Integrations AWS CLI *Command Reference*.

Step 7: Remove the managed thing from your hub

Clean up your hub by removing the managed thing.

To delete a managed thing

- Use the `delete-managed-thing` command to remove a managed thing from your device hub.

  delete-managed-thing example

  ```
  aws iot-managed-integrations delete-managed-thing \
  --identifier "DEVICE_MANAGED_THING_ID"
  ```

  For more information, see the delete-managed-thing command in the Managed Integrations AWS CLI *Command Reference*.

  > **Note**
  >
  > If the device is stuck in a `DELETE_IN_PROGRESS` state, append the `--force` flag to the `delete-managed-thing command`.

  > **Note**
  >
  > For Z-wave devices, you need to put the device into pairing mode after executing the command.

User guided setup to onboard and operate devices

Set up your devices to be onboarded to your Managed Integrations hub by creating a managed thing an connecting it to your hub. This section describes the steps to complete the device onboarding process using user guided setup.

Prerequisites

Complete these steps before attempting to onboard a device:

- Onboard a hub device to the Managed Integrations hub.
- Install the latest version of AWS CLI from the Managed Integrations AWS CLI Command Reference
- Subscribe to DEVICE_DISCOVERY_STATUS event notifications.

User guided setup steps

- Prerequisite: Enable pairing mode on your Z Wave device
- Step 1: Start device discovery
- Step 2: Query the discovery job ID
- Step 3: Create a managed thing for your device
- Step 4: Query the managed thing
- Step 5: Get managed thing capabilities
- Step 6: Send a command to the managed thing
- Step 7: Check the managed thing state
- Step 8: Remove managed thing from your hub

Prerequisite: Enable pairing mode on your Z Wave device

Enable pairing mode on the Z-wave device. The pairing mode can vary for each Z-Wave device, so refer to the device's instructions to properly set up the pairing mode. It is usually a button that the user must press.

Step 1: Start device discovery

Start device discovery for your hub to obtain a discovery job ID which is used to onboard your device.

To start device discovery

- Use the start-device-discovery command to obtain the discovery job ID.

  start-device-discovery example

```
#For Zigbee
  aws iot-managed-integrations start-device-discovery --discovery-type ZIGBEE \
  --controller-identifier HUB_MANAGED_THING_ID
```

```
#For Zwave
aws iot-managed-integrations start-device-discovery --discovery-type ZWAVE \
--controller-identifier HUB_MANAGED_THING \
--authentication-material-type ZWAVE_INSTALL_CODE \
--authentication-material 13333

#For Cloud
aws iot-managed-integrations start-device-discovery --discovery-type CLOUD \
--account-association-id C2C_ASSOCIATION_ID \

#For Custom
aws iot-managed-thing start-device-discovery --discovery-type CUSTOM \
--controller-identifier HUB_MANAGED_THING_ID \
--custom-protocol-detail NAME : NON_EMPTY_STRING \
```

Response:

```
{
      "Id": DISCOVERY_JOB_ID,
      "StartedAt": "2025-06-03T14:43:12.726000-07:00"
  }
```

> **Note**
>
> There are separate commands for Z-wave and Zigbee devices.

For more information, see the start-device-discovery API in the Managed Integrations AWS CLI *Command Reference*.

Step 2: Query the discovery job ID

Use the `list-discovered-devices` command to get the authentication material of your device.

To query your discovery job ID

- Use the discovery job ID with the list-discovered-devices command to get the authentication material of your device.

```
aws iot-managed-integrations list-discovered-devices --identifier DISCOVERY_JOB_ID
```

Response:

```
"Items": [
    {
        "DeviceTypes": [],
        "DiscoveredAt": "2025-06-03T14:43:37.619000-07:00",
        "AuthenticationMaterial": AUTHENTICATION_MATERIAL
    }
]
```

Populate Brand and Model for discovered devices

During device discovery, discovered devices can optionally report brand and model information through the Matter Basic Information cluster. These values are then returned in the ListDiscoveredDevices and ListManagedThings responses.

Report the Basic Information cluster (cluster `0x0028`) on the device's root endpoint (`0`) and map:

- `VendorName` (attribute `0x0001`) → `Brand`
- `ProductName` (attribute `0x0003`) → `Model`

The following example shows how to report the Basic Information cluster on the root endpoint:

```
"endpoints": [{
    "id": "0",
    "clusters": [{
        "id": "0x0028",
        "attributes": [{
            "id": "0x0001",
            "value": "ExampleBrand"
        }, {
            "id": "0x0003",
            "value": "ExampleModel-X1"
        }]
    }]
}]
```

Both values are strings with a maximum length of 32 characters. If the Basic Information cluster or either attribute is not reported, the corresponding field is omitted from the response.

Step 3: Create a managed thing for your device

Use the `create-managed-thing` command to create a managed thing for your device. Each device requires its own managed thing.

To create a managed thing

- Use the `create-managed-thing` command to create a managed thing for your device.

  create-managed-thing example

  ```
  aws iot-managed-integrations create-managed-thing \
    --role  DEVICE --authentication-material-type DISCOVERED_DEVICE \
    --authentication-material "AUTHENTICATION_MATERIAL"
  ```

  Response:

  ```
  {
      "Id": "DEVICE_MANAGED_THING_ID"
      "Arn": "arn:aws:iotmanagedintegrations:AWS_REGION:AWS_ACCOUNT_ID:managed-
  thing/DEVICE_MANAGED_THING_ID"
      "CreatedAt": "2025-06-09T13:58:52.977000+08:00"
    }
  ```

  For more information, see the create-managed-thing command in the Managed Integrations AWS CLI *Command Reference*.

Step 4: Query the managed thing

You can check if a managed thing is activated by using the `get-managed-thing` command.

To query a managed thing

- Use the `get-managed-thing` command to check if the managed thing's provisioning status is set to `ACTIVATED`. For more information on provisioning status, see Device Provisioning.

  get-managed-thing example

```
aws iot-managed-integrations get-managed-thing \
  --identifier "DEVICE_MANAGED_THING_ID"
```

Response:

```
{
      "Id": "DEVICE_MANAGED_THING_ID",
      "Arn": "arn:aws:iotmanagedintegrations:AWS_REGION:AWS_ACCOUNT_ID:managed-
thing/DEVICE_MANAGED_THING_ID",
      "Role": "DEVICE",
      "ProvisioningStatus": "ACTIVATED",
      "MacAddress": "MAC_ADDRESS",
      "ParentControllerId": "PARENT_CONTROLLER_ID",
      "CreatedAt": "2025-06-03T14:46:35.149000-07:00",
      "UpdatedAt": "2025-06-03T14:46:37.500000-07:00",
      "Tags": {}
  }
```

For more information, see the get-managed-thing command in the Managed Integrations AWS CLI *Command Reference*.

Step 5: Get managed thing capabilities

You can view a list of a managed thing's available actions by using the `get-managed-thing-capabiltiies`.

To get a device's capabilities

- Use the `get-managed-thing-capabilities` command to obtain the endpoint ID. Also note the list of possible actions.

  get-managed-thing-capabilities example

```
aws iot-managed-integrations get-managed-thing-capabilities \
  --identifier "DEVICE_MANAGED_THING_ID"
```

  Response:

```
{
```

```
    "ManagedThingId": "DEVICE_MANAGED_THING_ID",
    "CapabilityReport": {
        "version": "1.0.0",
        "nodeId": "zb.539D+4A1D",
        "endpoints": [
            {
                "id": "1",
                "deviceTypes": [
                    "Unknown Device"
                ],
                "capabilities": [
                    {
                        "id": "matter.OnOff@1.4",
                        "name": "On/Off",
                        "version": "6",
                        "properties": [
                            "OnOff",
                            "OnOff",
                            "OnTime",
                            "OffWaitTime"
                        ],
                        "actions": [
                            "Off",
                            "On",
                            "Toggle",
                            "OffWithEffect",
                            "OnWithRecallGlobalScene",
                            "OnWithTimedOff"
                        ],
                        ...
}
```

For more information, see the get-managed-thing-capabilities command in the Managed Integrations AWS CLI*Command Reference*.

Step 6: Send a command to the managed thing

You can use the send-managed-thing-command command to send a toggle action command to your managed thing.

Send a command to the managed thing using a toggle action.

- Use the `send-managed-thing-command` command to send a toggle action command.

  send-managed-thing-command example

```
json=$(jq -cr '.|@json') <<EOF
  [
    {
      "endpointId": "1",
      "capabilities": [
        {
          "id": "matter.OnOff@1.4",
          "name": "On/Off",
          "version": "1",
          "actions": [
            {
              "name": "Toggle",
              "parameters": {}
            }
          ]
        }
      ]
    }
  ]
  EOF
  aws iot-managed-integrations send-managed-thing-command \
  --managed-thing-id ${device_managed_thing_id} --endpoints ENDPOINT_ID
```

> **Note**
>
> This example uses jq cli to but you can also pass the entire `endpointId` string

Response:

```
{
  "TraceId": TRACE_ID
  }
```

For more information, see the send-managed-thing-command command in the Managed Integrations AWS CLI *Command Reference*.

Step 7: Check the managed thing state

Check the managed thing's state to validate the toggle action succeeded.

To check a managed thing's device state

- Use the `get-managed-thing-state` command to validate the toggle action succeeded.

  get-managed-thing-state example

```
  aws iot-managed-integrations get-managed-thing-state --managed-thing-
id DEVICE_MANAGED_THING_ID
```

  Response:

```
{
      "Endpoints": [
          {
              "endpointId": "1",
              "capabilities": [
                  {
                      "id": "matter.OnOff@1.4",
                      "name": "On/Off",
                      "version": "1.4",
                      "properties": [
                          {
                              "name": "OnOff",
                              "value": {
                                  "propertyValue": true,
                                  "lastChangedAt": "2025-06-03T21:50:39.886Z"
                              }
                          }
                      ]
                  }
              ]
          }
      ]
```

```
}
```

For more information, see the get-managed-thing-state command in the Managed Integrations AWS CLI *Command Reference*.

Step 8: Remove managed thing from your hub

Clean up your hub by removing the managed thing.

To delete a managed thing

- Use the delete-managed-thing command to remove a managed thing.

  delete-managed-thing example

  ```
  aws iot-managed-integrations delete-managed-thing \
    --identifier MANAGED_THING_ID
  ```

  For more information, see the delete-managed-thing command in the Managed Integrations AWS CLI *Command Reference*.

  > **Note**
  >
  > If the device is stuck in a `DELETE_IN_PROGRESS` state, append the `--force` flag to the `delete-managed-thing` command.

  > **Note**
  >
  > For Z-wave devices, you need to put the device into pairing mode after executing the command.

Capability rediscovery

This section describes the steps to relearn device capability information for hub-connected devices using capability rediscovery.

Prerequisites

Complete these steps before attempting capability rediscovery:

- Onboard a hub device to the Managed Integrations hub.
- Onboard one or more end devices to the hub.
- Install the latest version of AWS CLI from the Managed Integrations AWS CLI Command Reference
- (Optional) Subscribe to DEVICE_DISCOVERY_STATUS event notifications.

When to use capability rediscovery

Use capability rediscovery in the following scenarios:

- After a firmware update to an end device that adds or modifies device capabilities
- After a hub software update that enables support for new device features
- When device capabilities are not accurately reflected in Managed Integrations
- To refresh the capability information for a single device or all devices connected to a hub

Start capability rediscovery

Start capability rediscovery for your hub to update device capabilities.

To start capability rediscovery

- Use the start-device-discovery command with the `CONTROLLER_CAPABILITY_REDISCOVERY` discovery type.

  start-device-discovery example

```
# For a single device
aws iot-managed-integrations start-device-discovery \
  --discovery-type CONTROLLER_CAPABILITY_REDISCOVERY \
  --controller-identifier HUB_MANAGED_THING_ID \
  --protocol PROTOCOL \
  --end-device-identifier DEVICE_MANAGED_THING_ID

# For all devices on a protocol
```

```
aws iot-managed-integrations start-device-discovery \
  --discovery-type CONTROLLER_CAPABILITY_REDISCOVERY \
  --controller-identifier HUB_MANAGED_THING_ID \
  --protocol PROTOCOL
```

Response:

```
{
    "Id": "a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6",
    "StartedAt": "2026-01-12T23:07:49.275Z"
}
```

> **Note**
>
> The `--protocol` parameter accepts `ZIGBEE`, `ZWAVE`, or `CUSTOM`.
> If the optional `--end-device-identifier` parameter is not provided, all devices associated with the hub for the selected protocol are rediscovered.

For more information, see the start-device-discovery command in the Managed Integrations AWS CLI *Command Reference*.

To verify that the device capabilities have been updated, see Get managed thing capabilities.

Troubleshooting

Use the following guidance to troubleshoot common issues with capability rediscovery. When capability rediscovery is in progress, Managed Integrations sends notifications through Kinesis or EventBridge as JSON messages.

Success notifications

Per device being processed:

```
{
    "messageType": "DEVICE_DISCOVERY_STATUS",
    // Capability rediscovery with controller and end device
    "resources": ["arn:aws:iotmanagedintegrations:us-east-1:123456789012:managed-thing/
abc123def456", "arn:aws:iotmanagedintegrations:us-east-1:123456789012:managed-thing/
xyz789uvw012"],
```

```
    "payload": {
        "deviceDiscoveryId": "a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6",
        "status": "RUNNING",
        "deviceCount": 1
    }
}
```

Completion:

```
{
    "messageType": "DEVICE_DISCOVERY_STATUS",
    // Capability rediscovery with controller only
    "resources": ["arn:aws:iotmanagedintegrations:us-east-1:123456789012:managed-thing/
abc123def456"],
    "payload": {
        "deviceDiscoveryId": "a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6",
        "status": "SUCCEEDED",
        "deviceCount": 0
    }
}
```

Failure notifications

Device failure:

```
{
    "messageType": "DEVICE_DISCOVERY_STATUS",
    // Capability rediscovery with controller and end device
    "resources": ["arn:aws:iotmanagedintegrations:us-east-1:123456789012:managed-thing/
abc123def456", "arn:aws:iotmanagedintegrations:us-east-1:123456789012:managed-thing/
ghi345jkl678"],
    "payload": {
        "deviceDiscoveryId": "a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6",
        "status": "RUNNING",
        "deviceCount": 1,
        "error": {
            "code": "500",
            "message": "Device unreachable",
            "managedThingId": "ghi345jkl678"
        }
    }
}
```

Complete failure:

```
{
    "messageType": "DEVICE_DISCOVERY_STATUS",
    // Capability rediscovery with controller only
    "resources": ["arn:aws:iotmanagedintegrations:us-east-1:123456789012:managed-thing/
abc123def456"],
    "payload": {
        "deviceDiscoveryId": "a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6",
        "status": "FAILED",
        "deviceCount": 0,
        "error": {
            "code": "404",
            "message": "Specified controller does not exist"
        }
    }
}
```

No notifications received

If you do not receive notifications about the discovery status, verify that you have subscribed to `DEVICE_DISCOVERY_STATUS` event notifications. For more information on setting up notifications, see Set up Managed Integrations notifications.

Some devices not updated

If some devices fail to update during capability rediscovery, check the error messages in the `DEVICE_DISCOVERY_STATUS` notifications. Common error messages include:

- `Device unreachable` - The device is offline or not responding
- `Timeout querying device` - The device is not responding within the expected timeframe
- `Invalid capability report` - The device firmware may have an issue

To resolve these issues, ensure the affected devices are powered on and reachable, then retry capability rediscovery for the failed devices individually using the `--end-device-identifier` parameter.

Discovery never completes

If the capability rediscovery process does not complete, check the discovery status using the discovery job ID returned from the `start-device-discovery` command:

```
aws iot-managed-integrations get-device-discovery \
  --identifier DISCOVERY_JOB_ID
```

If the discovery is stuck:

- Verify that the hub is online and connected to Managed Integrations
- Wait up to 15 minutes for the discovery to timeout automatically
- Retry the capability rediscovery after confirming the hub and devices are operational

WiFi Simple Setup to onboard and operate devices

WiFi Simple Setup (WSS) is an automated device onboarding method that simplifies WiFi credential provisioning for IoT devices managed through AWS IoT Managed Integrations.

On this page:

- the section called “What is WiFi Simple Setup”
- the section called “Common use cases”
- the section called “When to use WSS”
- the section called “Prerequisites”
- the section called “How WSS works”
- the section called “WSS workflow”
- the section called “Deployment scenarios”

What is WiFi Simple Setup

WiFi Simple Setup (WSS) enables devices to automatically receive WiFi credentials from a provisioner device (such as a hub) through a secure, automated process. After a one-time barcode scan, the entire WiFi connection process completes automatically without requiring end-users to manually enter WiFi passwords or select networks.

Key characteristics

- One-time barcode scan to activate device
- Automatic discovery and connection
- Secure local credential exchange using TLS 1.2/1.3

- Configurable time-bounded activation window (Default 15 mins)
- Automatic fallback to User Guided Setup if needed

Common use cases

- Smart home cameras requiring WiFi connectivity
- IoT sensors in residential environments
- WiFi-enabled appliances and devices
- Any managed integrations device requiring automated WiFi setup

Example scenario

A customer purchases a smart home camera. After unboxing, they scan the device barcode with their mobile app, power on the camera, and within 60 seconds the camera automatically connects to their WiFi network without manual password entry. As part of the same process, the camera is also onboarded with managed integrations and can now be controlled using managed integrations APIs.

When to use WSS

Comparison of onboarding methods

Onboarding method comparison

Method	User Action	Best For	Automation Level
WSS	Scan barcode	WiFi devices needing automated setup	High - automatic after scan
SS (Simple Setup)	Scan QR + manual pairing	Protocol-specific devices (Zigbee, Z-Wave)	Medium - requires pairing steps
ZTS (Zero Touch)	None (pre-registered)	Enterprise deployments with fulfillment integrati on	Highest - fully automatic

Method	User Action	Best For	Automation Level
UGS (User Guided)	Button presses + manual steps	Fallback when automation fails	Low - manual intervention

When to choose WSS

- Device requires WiFi connectivity
- Hub or provisioner available in household
- Streamlined setup experience desired
- Mobile app has barcode scanning capability

When to use alternatives

- **ZTS:** Enterprise deployments with fulfillment center pre-registration
- **SS:** Protocol-specific devices (Zigbee, Z-Wave) with different pairing requirements
- **UGS:** Fallback when WSS unavailable or fails

Prerequisites

For provisioner devices (hubs)

- Hub SDK integration with WiFi connectivity
- Software access point (SoftAP) creation capability
- Access to local WiFi credentials via customer-provided API
- Registered as managed integrations CONTROLLER role with credential locker

For provisionee devices

- Managed integrations End device SDK integration with WiFi capability
- Hardware security module (HSM) or Trusted Platform Module (TPM)
- Claim certificate and private key securely stored
- Unique Serial Number (SN: 12-50 characters) and UPC/EAN
- Barcode labels on device or packaging

> **Note**
>
> **EAN Support:** Provisioners currently support UPC only. EAN support is planned for future releases.

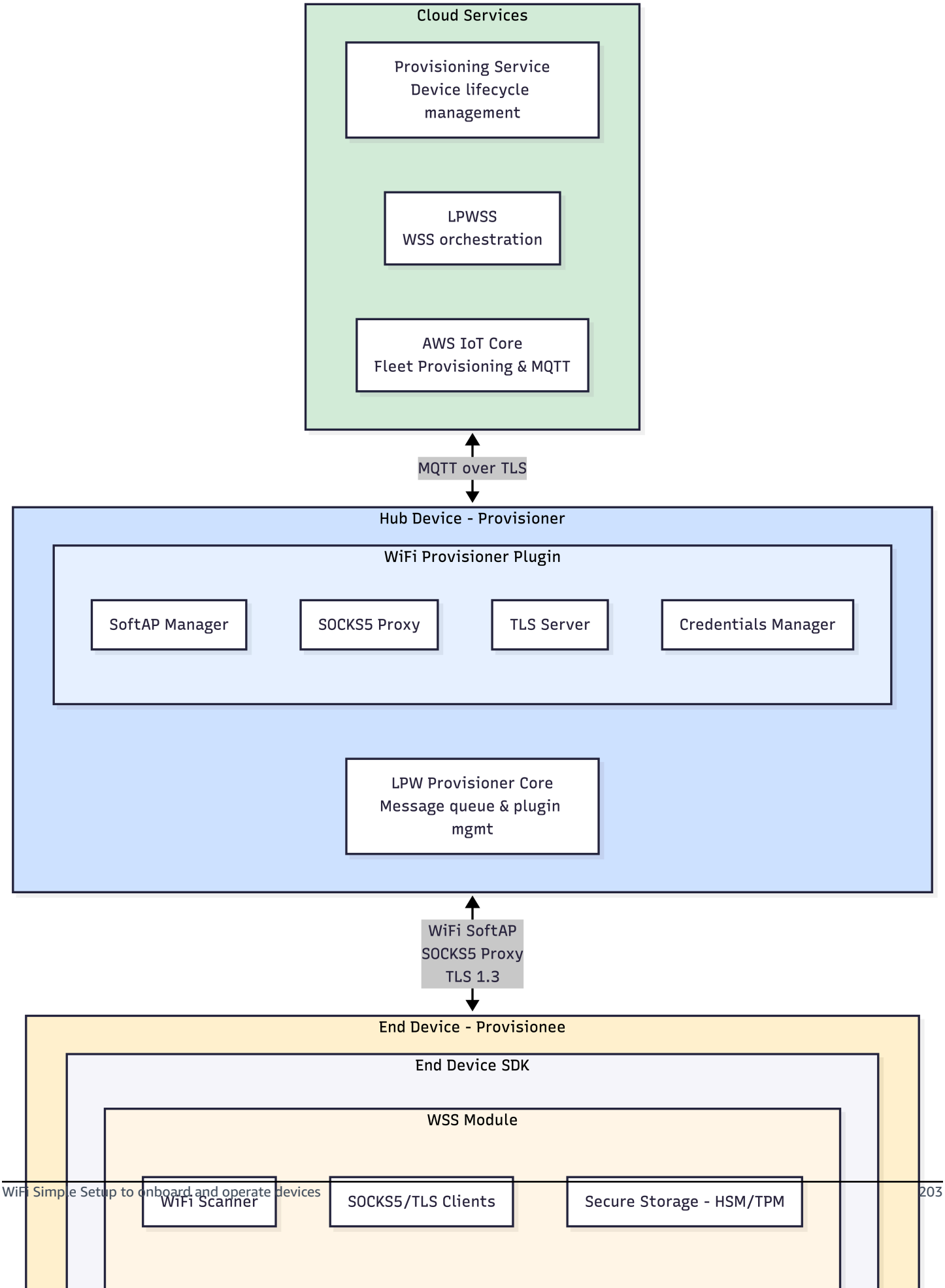
For customer implementation

- Managed integrations account configured
- Fleet Provisioning setup (custom endpoint, provisioning profile, template)
- Mobile application with barcode scanning capability
- Customer API for WiFi credential access

How WSS works

Architecture overview

The following diagram shows the WSS architecture with cloud services, provisioner hub, and provisionee device components:


Cloud Services
Provisioning Service
Device lifecycle
management
LPWSS
WSS orchestration
AWS IoT Core
Fleet Provisioning & MQTT
MQTT over TLS
Hub Device - Provisioner
WiFi Provisioner Plugin
SoftAP Manager
SOCKS5 Proxy
TLS Server
Credentials Manager
LPW Provisioner Core
Message queue & plugin
mgmt
WiFi SoftAP
SOCKS5 Proxy
TLS 1.3
End Device - Provisionee
End Device SDK
WSS Module
WiFi Scanner
SOCKS5/TLS Clients
Secure Storage - HSM/TPM

Key components

Cloud services: Coordinate authentication, manage device lifecycle, and distribute session tokens for secure credential exchange.

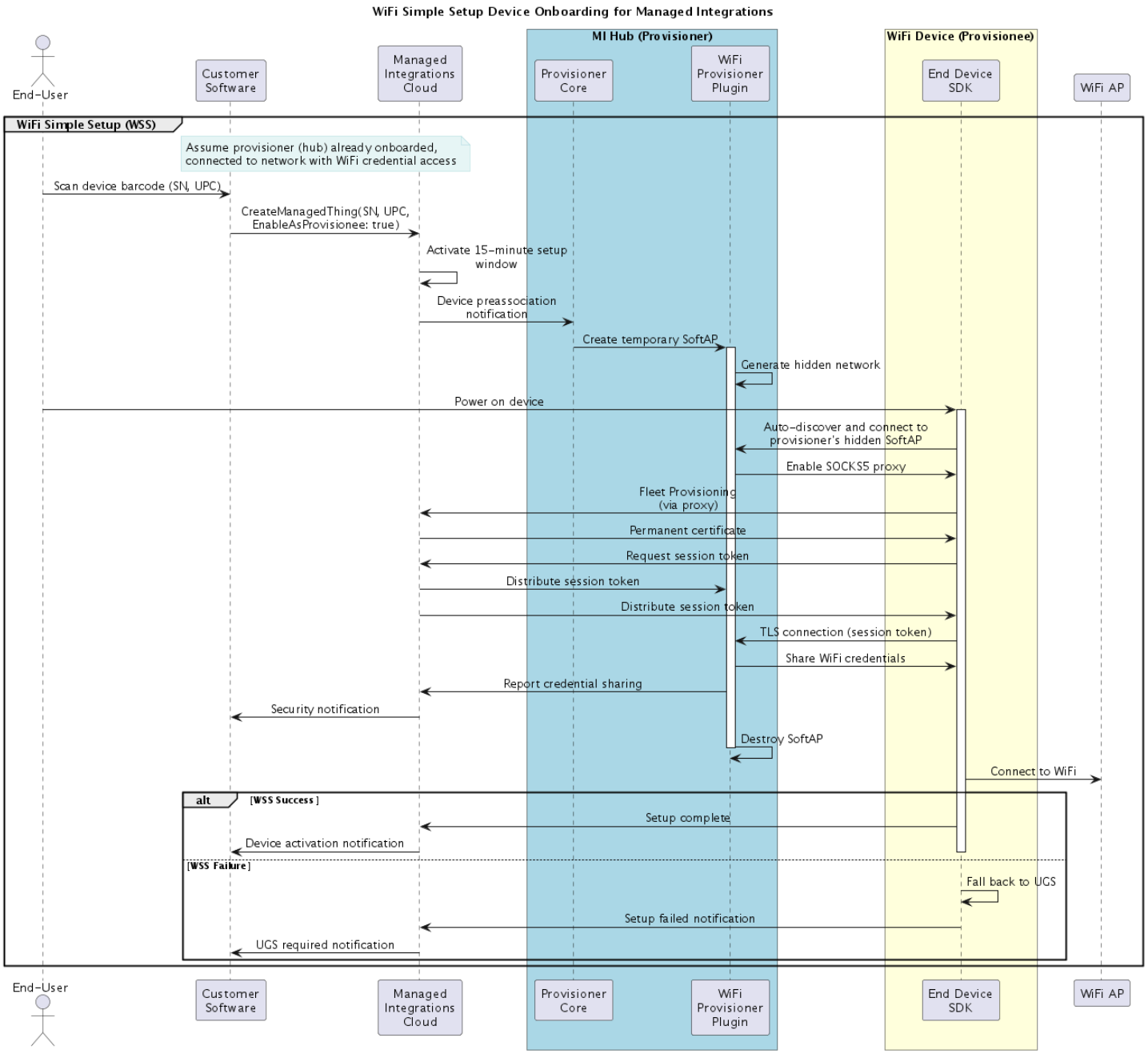
Provisioner (Hub): WiFi-connected device that creates temporary access point and shares WiFi credentials with new devices.

Provisionee (Device): New WiFi device requiring network access for initial setup and operation.

Mobile application: Customer-provided app that initiates setup via barcode scanning.

WSS workflow

The following diagram shows the complete WiFi Simple Setup workflow from barcode scanning through device activation:



Workflow phases

Phase 1: Account linking

End-user scans device barcode (SN + UPC), activating a 15-minute setup window. Cloud notifies all eligible provisioners in the household.

> **Important**
>
> Only one provisionee can be onboarded at a time. If you scan multiple devices at a time, only the latest one will be onboarded. If you want to onboard devices that were already scanned, you need to run `UpdateManagedThing`.

Phase 2: Device discovery

Device powers on, calculates temporary credentials, and automatically connects to provisioner's hidden temporary network.

Phase 3: Cloud authentication

Device completes Fleet Provisioning via provisioner's restricted proxy, obtaining permanent certificate. Cloud validates device and provisioner relationship, then distributes session tokens.

Phase 4: Credential exchange

Device establishes secure TLS connection to provisioner using session token. Provisioner shares WiFi credentials. Provisioner reports credential sharing for security monitoring.

Phase 5: Network connection

Device connects to WiFi network and reports success to cloud. Setup complete—device is operational.

Fallback: If any phase fails, device automatically falls back to User Guided Setup with mobile app guidance.

Deployment scenarios

Scenario 1: Standard hub with WiFi access

Hub connected to WiFi with access to credentials via customer API. Hub shares credentials directly with provisionee without cloud WiFi storage.

Scenario 2: Multiple provisioners

Multiple hubs in household provide redundancy. First provisioner to respond serves the device. Automatic load distribution improves reliability.

Scenario 3: Automatic fallback

If provisioner unavailable or connection fails, device automatically falls back to User Guided Setup. Mobile app guides user through manual setup. Fallback is transparent to end-user.

Configure provisioners for WiFi Simple Setup

On this page:

- the section called “Provisioner requirements”
- the section called “Enable provisioner capability”
- the section called “Provisioner setup workflow”
- the section called “WiFi credential access”
- the section called “Verify provisioner status”
- the section called “Disable provisioner capability”

Provisioner requirements

Provisioner devices must meet these requirements to support WiFi Simple Setup.

Hardware and connectivity

- Hub SDK integration completed
- WiFi connectivity to network
- Software access point (SoftAP) creation capability
- Sufficient system resources for temporary network services

Software and security

- Access to local WiFi credentials via customer-provided API
- Ability to create restricted SOCKS5 proxy (port 1080)
- TLS 1.2/1.3 with Pre-Shared Key (PSK) support for credential exchange (port 4433)

Registration requirements

- Registered as managed integrations CONTROLLER role
- Associated with a credential locker

- Device status: DISCOVERED or ACTIVATED

Enable provisioner capability

Enable WSS provisioner functionality using CreateManagedThing or UpdateManagedThing APIs.

For new devices

```
{
  "role": "CONTROLLER",
  "credentialLockerId": "ad5cdc9f786b4dbe9490e57c0b1d900e",
  "authenticationMaterialType": "WIFI_SETUP_QR_BAR_CODE",
  "authenticationMaterial": "SN:12345679012;UPC:987654321012",
  "wiFiSimpleSetupConfiguration": {
    "enableAsProvisioner": true
  }
}
```

For existing devices

```
{
  "managedThingId": "existing-hub-id",
  "wiFiSimpleSetupConfiguration": {
    "enableAsProvisioner": true
  }
}
```

Parameters:

- `enableAsProvisioner`: Boolean flag enabling WSS provisioner capability (default: false)

Provisioner setup workflow

Configure a hub as WSS provisioner following these steps:

Step 1: Register and enable

Register hub with CreateManagedThing API:

- Set role as "CONTROLLER"
- Specify credential locker ID for household association
- Include `wiFiSimpleSetupConfiguration` with `enableAsProvisioner: true`
- Or use UpdateManagedThing for existing devices

Step 2: Configure WiFi credentials path

Configure the path to your WiFi credentials file in iotmi_config.json to enable the provisioner to access WiFi credentials (SSID and password) for sharing with new devices.

Add the `wss_local_wifi_cred_path` parameter to the `ro` (read-only) section of your iotmi_config.json file:

```
{
  "ro": {
    "wss_local_wifi_cred_path": "/etc/wpa_supplicant/wpa_supplicant-wlan0.conf"
  }
}
```

Parameters:

- `wss_local_wifi_cred_path`: File path to the wpa_supplicant configuration file containing WiFi credentials

This configuration is required for the Hub SDK to access your WiFi credentials during the provisioning process.

Step 3: Capability reporting

Hub SDK automatically evaluates and reports capabilities:

- Hub assesses WiFi credential access
- Hub validates SoftAP creation capability
- Hub reports `supportAsProvisioner` status to cloud
- Cloud validates hub can function as provisioner

Step 4: Activation

Cloud confirms provisioner status:

- Hub becomes ready to receive provisioning requests
- Hub receives notifications for pending devices in same credential locker
- Hub can begin provisioning new devices

WiFi credential access

Provisioners must provide access to WiFi credentials for sharing with provisionee devices.

Implementation requirements

Implement a customer API that returns WiFi SSID and password:

- API accessible by Hub SDK components
- Read from system configuration files (e.g., `/etc/wpa_supplicant/wpa_supplicant.conf`)
- Handle appropriate file permissions

Security controls

Access restrictions:

- SSH access from SoftAP disabled by default
- WiFi credentials transmitted only over TLS 1.2/1.3 with PSK (port 4433)
- Session tokens are cryptographically secure (256-bit entropy), device-pair specific, single-use (5-minute validity)
- Session tokens distributed via cloud MQTT messaging for mutual authentication
- Credential access logged for security audit

Implementation approaches:

- Direct file system access with appropriate permissions
- System API calls for network configuration
- Custom credential management service with secure retrieval

Verify provisioner status

Monitor two configuration states to ensure proper provisioner functionality.

Configuration states

WSS provisioners have two critical state values that determine functionality:

Provisioner configuration states

State	Description	Source	Purpose
enableAsProvisioner	Customer's configuration intent	Set via CreateManagedThing/ UpdateManagedThing API	Enables WSS provisioner role for device
supportAsProvisioner	Actual device capability	Self-reported by hub after capability assessment	Indicates if device can technically function as provisioner

Key distinction:

- `enableAsProvisioner=true` + `supportAsProvisioner=false` = WSS will not work (device configured but incapable)
- `enableAsProvisioner=true` + `supportAsProvisioner=true` = WSS ready and functional
- Both must be true for successful provisioning operations

Default values:

- `enableAsProvisioner`: false (must be explicitly enabled)
- `supportAsProvisioner`: true for hub controllers, false for standard WiFi devices (determined by device assessment)

Troubleshooting: If `enableAsProvisioner=true` but `supportAsProvisioner=false`, check WiFi credential access and SoftAP capability.

Status notifications

Supported:

```
{
  "notificationType": "WSS_SUPPORTED",
  "managedThingId": "hub-device-id",
  "timestamp": "2025-06-04T20:22:04Z",
  "message": "Device can function as WSS provisioner"
}
```

Not supported:

```
{
  "notificationType": "WSS_NOT_SUPPORTED",
  "managedThingId": "hub-device-id",
  "timestamp": "2025-06-04T20:22:04Z",
  "reason": "NO_WIFI_CREDENTIAL_ACCESS",
  "message": "Device cannot function as WSS provisioner"
}
```

Common unsupported reasons

- No WiFi credential access configured
- Customer WiFi API not implemented
- Cannot create SoftAP
- Ethernet-only connection without WiFi radio
- Insufficient system resources

Verification steps

1. **Call GetManagedThing API** to check configuration and status:

```
{
```

```
  "managedThingId": "hub-device-id",
  "role": "CONTROLLER",
  "credentialLockerId": "ad5cdc9f786b4dbe9490e57c0b1d900e",
  "authenticationMaterialType": "WIFI_SETUP_QR_BAR_CODE",
  "authenticationMaterial": "SN:12345679012;UPC:987654321012",
  "wiFiSimpleSetupConfiguration": {
    "enableAsProvisioner": true,
    "supportAsProvisioner": true,
    "enableAsProvisionee": false,
    "wssExpirationTimeStamp": null
  }
}
```

Key fields:

- `enableAsProvisioner`: Customer configuration setting
- `supportAsProvisioner`: Device-reported capability status
- `wssExpirationTimeStamp`: Present only for provisionee devices with active WSS window

2. **Review cloud notifications** for capability reporting
3. **Verify WiFi credential API** returns valid credentials
4. **Check hub logs** for capability assessment details

Disable provisioner capability

Disable WSS provisioner functionality when no longer needed. Using UpdateManagedThing API.

```
{
  "managedThingId": "hub-device-id",
  "wiFiSimpleSetupConfiguration": {
    "enableAsProvisioner": false
  }
}
```

Impact of disabling

- Cloud updates `enableAsProvisioner` to false
- Hub stops receiving new provisioning notifications

- Pending provisioning operations cancelled
- Existing connections complete normally
- Hub no longer creates SoftAPs for new devices

Common use cases

- Security concerns or suspicious activity detected
- Hub being relocated or decommissioned
- Temporary maintenance or troubleshooting
- Customer preference changes

Note: Disabling provisioner does not affect hub's other functionality or existing provisionee devices.

Configure provisionees for WiFi Simple Setup

On this page:

- the section called “Provisionee requirements”
- the section called “Manufacturing requirements”
- the section called “Enable provisionee capability”
- the section called “Barcode scanning workflow”
- the section called “Activation window and timeouts”
- the section called “Retry WSS setup”
- the section called “Disable provisionee capability”

Provisionee requirements

Provisionee devices must meet these requirements to support WiFi Simple Setup.

Hardware and security

- Managed integrations End device SDK integration completed with build flag `IOTMI_USE_WSS_PROVISIONEE` enabled

- Hardware security module (HSM) or Trusted Platform Module (TPM) for secure storage
- WiFi connectivity capability
- Sufficient processing power for cryptographic operations (SHA-384, HKDF)

Authentication materials

- Claim certificate and private key stored securely in HSM/TPM
- Unique Serial Number (SN): 12-50 characters alphanumeric
- Universal Product Code (UPC): 12 digits OR European Article Number (EAN): 13 digits
- Barcode labels displaying SN and UPC/EAN (on device or packaging)

> **Note**
>
> **EAN Support:** Provisioners currently support UPC only. EAN support is planned for future releases. Provisionees support both UPC and EAN.

Important: SN paired with UPC/EAN must be unique per managed thing within customer AWS account.

Software components

- corePKCS11 Platform Abstraction Layer (PAL) implementation connecting PKCS#11 API to HSM/TPM
- TLS 1.2/1.3 client with Pre-Shared Key (PSK) capability (port 4433, cipher suite: TLS_AES_256_GCM_SHA384 or equivalent)
- SOCKS5 proxy client support (port 1080)
- Cryptographic functions for SHA-384 and HKDF

Note: corePKCS11 PAL must interface with actual HSM/TPM hardware, not software-only implementation.

Manufacturing requirements

Device manufacturers must provision secure materials and identifiers during manufacturing.

Secure material provisioning

Claim certificate and private key:

- Generate during manufacturing process
- Store in HSM or TPM (required for production devices)
- Never expose private key outside secure storage
- Access via corePKCS11 PAL interface

Device identifiers:

- Serial Number (SN): 12-50 character unique identifier
- Universal Product Code (UPC): 12-digit product code OR European Article Number (EAN): 13-digit product code
- Store securely alongside claim certificate in HSM/TPM

Barcode requirements

Physical labels must include:

- SN barcode on device or packaging (Code 128 format recommended)
- UPC/EAN barcode on device or packaging (UPC-A or EAN-13 format)
- Easily scannable labels (well-lit, focused, unobstructed)
- Labels that survive shipping and handling

Fleet Provisioning setup

AWS account configuration:

1. Create custom endpoint using GetCustomEndpoint API
2. Create provisioning profile using CreateProvisioningProfile API
3. Obtain claim certificate and private key for device family
4. Configure Fleet Provisioning template with required fields:
   - `deviceSerialNumber` (SN)
   - `universalProductCode` (UPC) or `europeanArticleNumber` (EAN)

- Device certificate generation support
- Thing registration in AWS IoT Core

Enable provisionee capability

Enable WSS for devices using CreateManagedThing API during account linking.

Device registration

```
{
  "role": "DEVICE",
  "credentialLockerId": "ad5cdc9f786b4dbe9490e57c0b1d900e",
  "authenticationMaterialType": "WIFI_SETUP_QR_BAR_CODE",
  "authenticationMaterial": "SN:123456789331;UPC:123456789331",
  "wiFiSimpleSetupConfiguration": {
    "enableAsProvisionee": true,
    "timeoutInMinutes": 15
  }
}
```

Parameters:

- `role`: Must be "DEVICE" for provisionee devices
- `credentialLockerId`: Associates device with household (same as provisioner)
- `authenticationMaterialType`: Use "WIFI_SETUP_QR_BAR_CODE" for WSS
- `authenticationMaterial`: SN and UPC/EAN from device barcodes
- `enableAsProvisionee`: Set to true to activate WSS
- `timeoutInMinutes`: Activation window duration (5-15 minutes, default 15)

Retry or reactivate

Use UpdateManagedThing to retry after initial failure:

```
{
  "managedThingId": "existing-device-id",
  "wiFiSimpleSetupConfiguration": {
```

```
    "enableAsProvisionee": true,
    "timeoutInMinutes": 15
  }
}
```

Barcode scanning workflow

Complete end-to-end workflow from barcode scanning to device activation.

Step 1: Preparation

- Ensure mobile application installed and authenticated
- Verify provisioner device (hub) is powered on and connected
- Have device and packaging accessible for scanning

Step 2: Scan and register

1. Open mobile application and navigate to device setup
2. Scan device Serial Number (SN) barcode
3. Scan Universal Product Code (UPC) or European Article Number (EAN) barcode
4. Mobile app validates scanned data
5. Mobile app calls CreateManagedThing API with scanned data and WSS configuration

> **Important**
>
> **Single provisionee onboarding limitation:** Only one provisionee can be onboarded at a time. If you scan multiple devices simultaneously, only the latest scanned device will be onboarded. To onboard devices that were previously scanned but not activated, you must run `UpdateManagedThing` to reactivate their setup window.

Step 3: Cloud activation

Cloud processing automatically:

1. Creates managed thing record with WSS configuration
2. Activates 15-minute setup window

3. Identifies eligible provisioners (same credential locker, `enableAsProvisioner=true`, `supportAsProvisioner=true`)
4. Sends provisioning notifications to all eligible provisioners
5. Provisioners create temporary SoftAPs and prepare for device connection

GetManagedThing response example:

```
{
  "managedThingId": "device-id",
  "role": "DEVICE",
  "credentialLockerId": "ad5cdc9f786b4dbe9490e57c0b1d900e",
  "authenticationMaterialType": "WIFI_SETUP_QR_BAR_CODE",
  "authenticationMaterial": "SN:123456789331;UPC:123456789331",
  "wiFiSimpleSetupConfiguration": {
    "enableAsProvisioner": false,
    "supportAsProvisioner": false,
    "enableAsProvisionee": true,
    "wssExpirationTimeStamp": "2025-06-04T20:22:04.069610Z"
  }
}
```

Key fields:

- `enableAsProvisionee`: Device configured for WSS provisioning
- `wssExpirationTimeStamp`: Active setup window end time (null if expired)

Step 4: Device power-on

1. Mobile app displays "Power on your device now" message
2. User powers on provisionee device
3. WSS workflow proceeds automatically:
   - Device discovers provisioner SoftAP
   - Authentication and credential exchange complete
   - Device connects to WiFi
   - Mobile app receives success notification

Activation window and timeouts

WSS uses time-limited windows and operation timeouts to enhance security and manage provisioning.

Activation window

Default: 15 minutes from barcode scan to device activation

Configuration:

- Set via `timeoutInMinutes` parameter (range: 5-15 minutes)
- Default: 15 minutes balances convenience and security

Expiration behavior:

- Provisioners remove device from allowlist after timeout
- Device cannot connect via WSS after expiration
- User must rescan barcodes or call UpdateManagedThing to reactivate

Operation timeouts

WSS operation timeouts

Operation	Timeout	Behavior
SoftAP discovery	30 seconds	Retries up to 5 times, then falls back to User Guided Setup
Fleet Provisioning	30 seconds	Retries with exponential backoff if fails
WiFi connection	60 seconds	Reports error and suggests troubleshooting if fails

Best practices

- Power on device within 5 minutes of barcode scan and place the device near provisioner's softap WiFi range

- Ensure provisioner is powered on and connected before scanning
- Use default 15-minute timeout unless specific requirements dictate otherwise

Retry WSS setup

If initial WSS setup fails, retry using UpdateManagedThing API.

Retry workflow

Step 1: Identify failure

- Mobile app receives WSS failure notification
- Timeout expires without successful activation
- User sees "Setup failed" or "Try again" message

Step 2: Retry via UpdateManagedThing

```
{
  "managedThingId": "device-managed-thing-id",
  "wiFiSimpleSetupConfiguration": {
    "enableAsProvisionee": true,
    "timeoutInMinutes": 15
  }
}
```

Step 3: Reactivation

- Cloud reactivates 15-minute setup window
- Provisioners recreate SoftAP and allowlist entry
- User powers on device (if not already powered on)
- WSS workflow proceeds

Common retry scenarios

WSS retry scenarios

Scenario	Solution
Device not powered on in time	Call UpdateManagedThing and power on device promptly
Provisioner unavailable	Ensure hub online, then retry
Network connectivity issues	Verify network stable, then retry
User error	Retry with correct device

Important: Each UpdateManagedThing call creates a new 15-minute window. After 3-4 failures, consider checking provisioner status, verifying device functionality, or using User Guided Setup.

Disable provisionee capability

Disable or permanently remove WSS for a provisionee device.

Using UpdateManagedThing API

```
{
  "managedThingId": "device-managed-thing-id",
  "wiFiSimpleSetupConfiguration": {
    "enableAsProvisionee": false
  }
}
```

Using DeleteManagedThing API

```
{
  "managedThingId": "device-managed-thing-id"
}
```

Comparison

Disable vs Delete comparison

Action	UpdateManagedThing	DeleteManagedThing
Device record	Retained in cloud	Permanently removed
Device status	Changed to disabled	No longer exists
Reactivation	Re-enable WSS anytime	Must re-register device
Use case	Temporary disable, testing	Device retired, security incident

Security considerations

Use disable/delete when:

- Suspicious activity detected
- Device reported stolen
- Security incident investigation required
- Device decommissioned or returned

After disabling/deleting, consider:

- Changing WiFi password if credentials may have been compromised
- Reviewing security notifications for unusual activity
- Checking other devices in household for similar issues

Create a custom certificate handler for secure storage

Device certificate management is crucial when onboarding the Managed Integrations hub. While certificates are stored in the file system by default, you can create a custom certificate handler for enhanced security and flexible credential management.

The managed integrations End device SDK provides a certificate handler to secure storage interface that you can implement as a shared object (.so) library. Build your secure storage implementation to read and write certificates, then link the library file to the HubOnboarding process at runtime.

API definition and components

Review the following `secure_storage_cert_handler_interface.hpp` file to understand the API components and requirements for your implementation

Topics

- API definition
- Key components

API definition

Contents of `secure_storage_cert_hander_interface.hpp`

```
/*
    * Copyright 2024 Amazon.com, Inc. or its affiliates. All rights reserved.
    *
    * AMAZON PROPRIETARY/CONFIDENTIAL
    *
    * You may not use this file except in compliance with the terms and
    * conditions set forth in the accompanying LICENSE.txt file.
    *
    * THESE MATERIALS ARE PROVIDED ON AN "AS IS" BASIS. AMAZON SPECIFICALLY
    * DISCLAIMS, WITH RESPECT TO THESE MATERIALS, ALL WARRANTIES, EXPRESS,
    * IMPLIED, OR STATUTORY, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY,
    * FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT.
    */
    #ifndef SECURE_STORAGE_CERT_HANDLER_INTERFACE_HPP
    #define SECURE_STORAGE_CERT_HANDLER_INTERFACE_HPP

    #include <iostream>
    #include <memory>

    namespace IoTManagedIntegrationsDevice {
    namespace CertHandler {
    /**
     * @enum CERT_TYPE_T
     * @brief enumeration defining certificate types.
```

```
     */
     typedef enum { CLAIM = 0, DHA = 1, PERMANENT = 2 } CERT_TYPE_T;
     class SecureStorageCertHandlerInterface {
      public:
       /**
        * @brief Read certificate and private key value of a particular certificate
        * type from secure storage.
        */
        virtual bool read_cert_and_private_key(const CERT_TYPE_T cert_type,
                                               std::string &cert_value,
                                               std::string &private_key_value) = 0;
        /**
          * @brief Write permanent certificate and private key value to secure storage.
          */
        virtual bool write_permanent_cert_and_private_key(
            std::string_view cert_value, std::string_view private_key_value) = 0;
        };
        std::shared_ptr<SecureStorageCertHandlerInterface>
createSecureStorageCertHandler();
   } //namespace CertHandler
   } //namespace IoTManagedIntegrationsDevice

   #endif //SECURE_STORAGE_CERT_HANDLER_INTERFACE_HPP
```

Key components

- CERT_TYPE_T - different types of certificates on the hub.
  - CLAIM - the claim cert originally on the hub, will be exchanged for a permanent cert.
  - DHA - unused for now.
  - PERMANENT - permanent cert to connect with Managed Integrations endpoint.
- read_cert_and_private_key - (FUNCTION TO BE IMPLEMENTED) Reads cert and key value in to the reference input. This function must be able to read both the CLAIM and PERMANENT cert, and is differentiated by the cert type mentioned above.
- write_permanent_cert_and_private_key - (FUNCTION TO BE IMPLEMENTED) writes permanent cert and key value to the desired location.

Example build

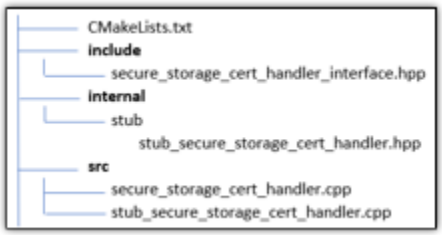
Separate your internal implementation headers from the public interface (`secure_storage_cert_handler_interface.hpp`) to maintain a clean project structure. With this separation, you can manage public and private components while building your certificate handler.

> **Note**
>
> Declare `secure_storage_cert_handler_interface.hpp` as public.

Topics

- Project structure
- Inherit the interface
- Implementation
- CMakeList.txt

Project structure



Inherit the interface

Create a concrete class that inherits the interface. Hide this header file and other files under a separate directory so that private and public headers can be differentiated easily when building.

```
#ifndef IOTMANAGEDINTEGRATIONSDEVICE_SDK_STUB_SECURE_STORAGE_CERT_HANDLER_HPP
  #define IOTMANAGEDINTEGRATIONSDEVICE_SDK_STUB_SECURE_STORAGE_CERT_HANDLER_HPP

  #include "secure_storage_cert_handler_interface.hpp"

  namespace IoTManagedIntegrationsDevice::CertHandler {
    class StubSecureStorageCertHandler : public SecureStorageCertHandlerInterface {
      public:
```

```
        StubSecureStorageCertHandler() = default;

        bool read_cert_and_private_key(const CERT_TYPE_T cert_type,
                                       std::string &cert_value,
                                       std::string &private_key_value) override;

        bool write_permanent_cert_and_private_key(
            std::string_view cert_value, std::string_view private_key_value) override;
            /*
            * any other resource for function you might need
            */


          };
      }
    #endif //IOTMANAGEDINTEGRATIONSDEVICE_SDK_STUB_SECURE_STORAGE_CERT_HANDLER_HPP
```

Implementation

Implement the storage class defined above, `src/stub_secure_storage_cert_handler.cpp`.

```
/*
  * Copyright 2024 Amazon.com, Inc. or its affiliates. All rights reserved.
  *
  * AMAZON PROPRIETARY/CONFIDENTIAL
  *
  * You may not use this file except in compliance with the terms and
  * conditions set forth in the accompanying LICENSE.txt file.
  *
  * THESE MATERIALS ARE PROVIDED ON AN "AS IS" BASIS. AMAZON SPECIFICALLY
  * DISCLAIMS, WITH RESPECT TO THESE MATERIALS, ALL WARRANTIES, EXPRESS,
  * IMPLIED, OR STATUTORY, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY,
  * FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT.
  */

  #include "stub_secure_storage_cert_handler.hpp"

  using namespace IoTManagedIntegrationsDevice::CertHandler;

  bool StubSecureStorageCertHandler::write_permanent_cert_and_private_key(
              std::string_view cert_value, std::string_view private_key_value) {
```

```
            // TODO: implement write function
            return true;
 }

 bool StubSecureStorageCertHandler::read_cert_and_private_key(const CERT_TYPE_T
cert_type,
                                                              std::string &cert_value,
                                                              std::string
&private_key_value) {
         std::cout<<"Using Stub Secure Storage Cert Handler, returning dummy values";
         cert_value = "StubCertVal";
         private_key_value = "StubKeyVal";
         // TODO: implement read function
         return true;
 }
```

Implement the factory function defined in the interface, `src/secure_storage_cert_handler.cpp`.

```
#include "stub_secure_storage_cert_handler.hpp"


 std::shared_ptr<IoTManagedIntegrationsDevice::CertHandler::SecureStorageCertHandlerInterface>
       IoTManagedIntegrationsDevice::CertHandler::createSecureStorageCertHandler() {
         // TODO: replace with your implementation
       return
 std::make_shared<IoTManagedIntegrationsDevice::CertHandler::StubSecureStorageCertHandler>();
     }
```

CMakeList.txt

```
#project name must stay the same
      project(SecureStorageCertHandler)

      # Public Header files. The interface definition must be in top level with exactly
 the same name
      #ie. Not in anotherDir/secure_storage_cert_hander_interface.hpp
      set(PUBLIC_HEADERS
```

```
                ${PROJECT_SOURCE_DIR}/include
        )

        # private implementation headers.
        set(PRIVATE_HEADERS
                ${PROJECT_SOURCE_DIR}/internal/stub
        )

        #set all sources
        set(SOURCES
                ${PROJECT_SOURCE_DIR}/src/secure_storage_cert_handler.cpp
                ${PROJECT_SOURCE_DIR}/src/stub_secure_storage_cert_handler.cpp
          )

        # Create the shared library
        add_library(${PROJECT_NAME} SHARED ${SOURCES})
        target_include_directories(
                ${PROJECT_NAME}
                PUBLIC
                    ${PUBLIC_HEADERS}
                PRIVATE
                    ${PRIVATE_HEADERS}
        )

        # Set the library output location. Location can be customized but version must
    stay the same
        set_target_properties(${PROJECT_NAME} PROPERTIES
                LIBRARY_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/../lib
                VERSION 1.0
                SOVERSION 1
        )

        # Install rules
        install(TARGETS ${PROJECT_NAME}
                LIBRARY DESTINATION lib
                ARCHIVE DESTINATION lib
        )

        install(FILES ${HEADERS}
                DESTINATION include/SecureStorageCertHandler
        )
```

Usage

After compilation, you'll have a `libSecureStorageCertHandler.so` shared object library file and its associated symbolic links. Copy both the library file and symbolic links to the library location expected by the HubOnboarding binary.

Topics

- Key considerations
- Use secure storage

Key considerations

- Verify that your user account has read and write permissions for both the HubOnboarding binary and `libSecureStorageCertHandler.so` library.
- Keep `secure_storage_cert_handler_interface.hpp` as your only public header file. All other header files should remain in your private implementation.
- Verify your shared object library name. While you build `libSecureStorageCertHandler.so`, HubOnboarding might require a specific version in the filename, such as `libSecureStorageCertHandler.so.1.0`. Use the `ldd` command to check library dependencies and create symbolic links as needed.
- If your implementation of the shared library has external dependencies, store them in a directory that HubOnboarding can access, such as `/usr/lib or the iotmi_common` directory.

Use secure storage

Update your `iotmi_config.json` file by setting both `iot_claim_cert_path` and `iot_claim_pk_path` to **`SECURE_STORAGE`**.

```
{
  "ro": {
    "iot_provisioning_method": "FLEET_PROVISIONING",
    "iot_claim_cert_path": "SECURE_STORAGE",
    "iot_claim_pk_path": "SECURE_STORAGE",
    "fp_template_name": "device-integration-example",
    "iot_endpoint_url": "[ACCOUNT-PREFIX]-ats.iot.AWS-REGION.amazonaws.com",
    "SN": "1234567890",
    "UPC": "1234567890"
```

```
  },
  "rw": {
    "iot_provisioning_state": "NOT_PROVISIONED"
  }
}
```

Custom protocol plugin

You can use a custom protocol plugin to integrate your proprietary IoT protocols into the managed integrations for AWS IoT Device Management ecosystem. Through well-defined SDK interfaces, you can onboard devices, define capabilities, and handle real-time control flows while maintaining full compatibility with managed integrations and hub SDK components.

The following lists discusses the key features of the custom protocol plugin.

Data model customization

Define your own AWS data model schemas and upload them to managed integrations during the provisioning flow. You can use these schemas later in your workflows.

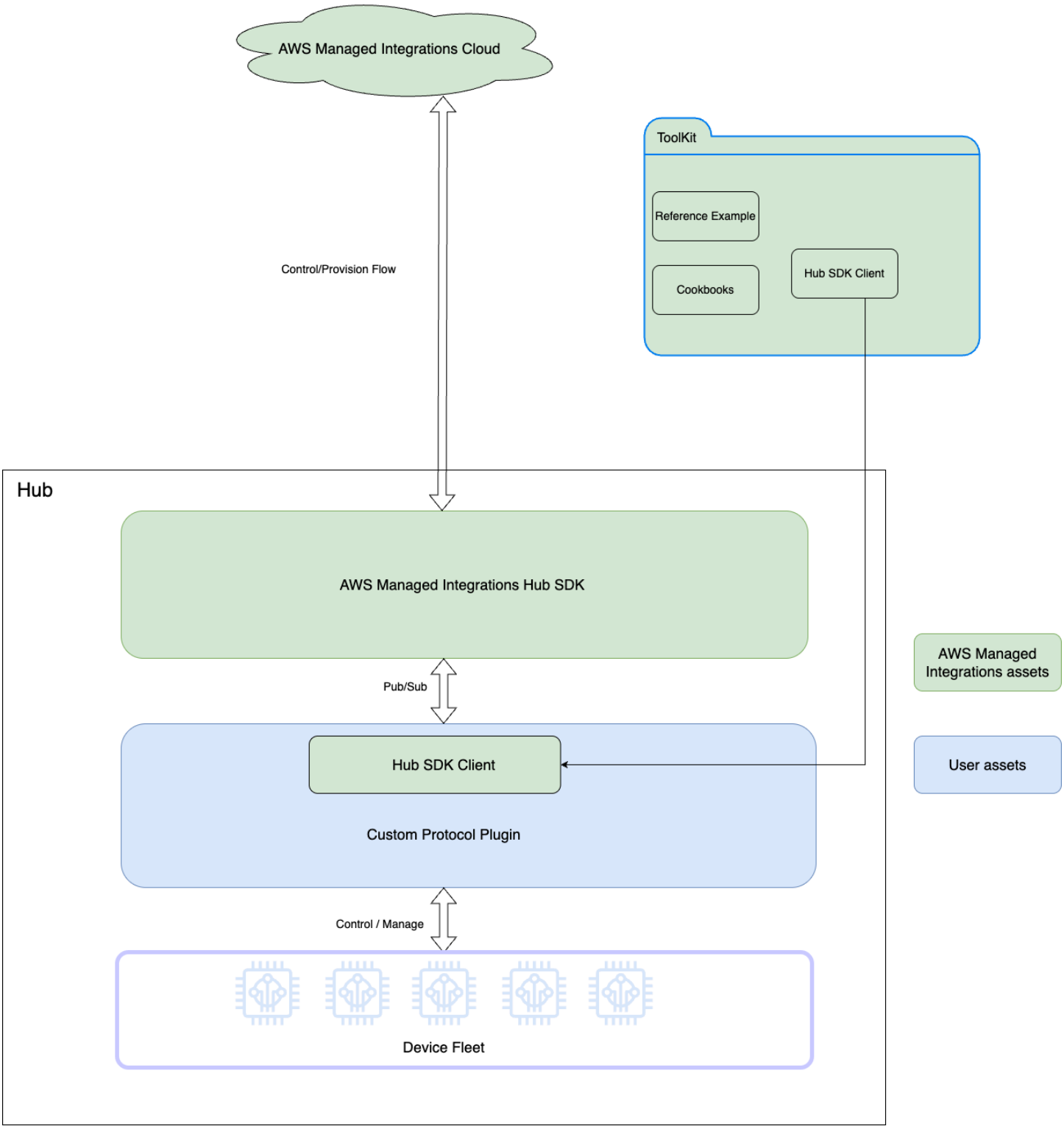
Flexible plugin implementation

- Build your own plugin component using Hub SDK client.
- Implement separate plugins for different functionalities, such as provisioning and control, or create a unified client for both.
- Maintain a clear boundary between managed integrations assets and your own assets such as middleware stacks, for decoupled and development-friendly code logic implementation.

Backward compatibility

For existing customers, smoothly onboard your new custom protocol while also keeping existing radio types working as-is.

The following diagram illustrates the custom protocol plugin architecture.



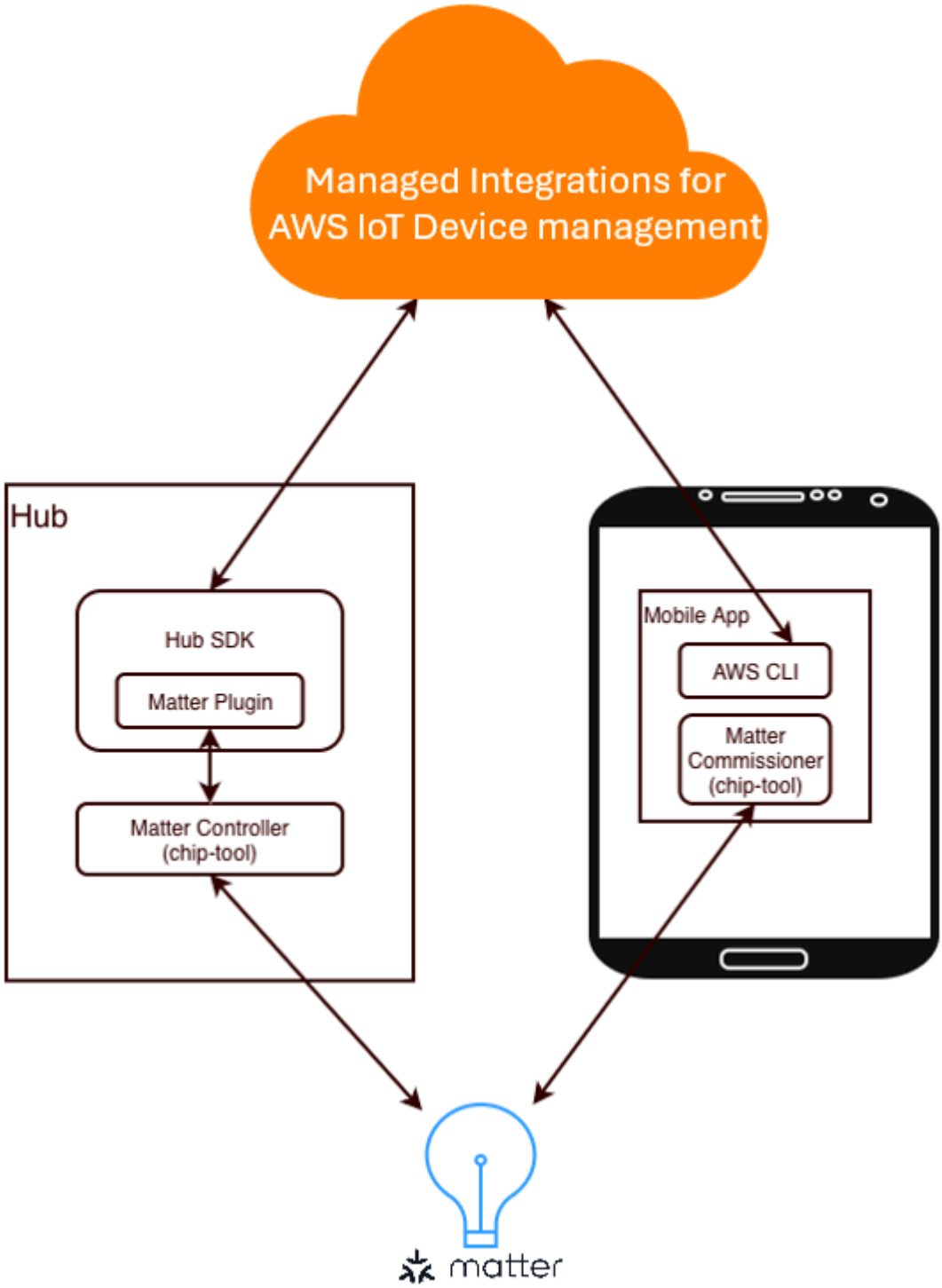
Matter Plugin

Topics

- What is Matter Plugin
- How to build Matter Plugin
- Quickstart: Set Up and Run the Matter Plugin
- The steps below are to be run on the Raspberry Pi i.e. the Commissioner.
- Supported Matter device types
- Adding Support for Additional Clusters
- Integrate different Matter Controller solutions

What is Matter Plugin

Matter Plugin is a reference implementation built using the Custom protocol plugin feature of the Managed integrations Hub SDK. It enables your hub to control Matter devices both locally on the same network, following the Matter specification, and remotely through Managed integrations.



The Matter Plugin is included in the Hub SDK. It communicates with Matter devices, implements Matter Controller functionalities, and exposes a remote control path via Managed integrations.

chip-tool is a command-line based reference controller from connectedhomeip. It acts as a Commissioner/Controller on a Matter fabric, allowing you to commission devices, read/write

attributes, and invoke commands and is intended for development and interoperability testing. In this guide, the Matter Plugin will use chip-tool to control the Matter device and also demonstrate Matter commissioning from the Mobile App.

The scope of this distribution does not include a production mobile application, nor does it cover certification and production activities such as CSA lab test and certification. These remain part of your product development process.

How to build Matter Plugin

The Matter Plugin has the following dependencies in addition to the Hub SDK:

- OpenSSL 3.0.x: The OpenSSL version requirement is not strict — in most cases, the system's default version works fine.
- nng v1.4.0
- cJSON v1.7.18
- AWS IoT Device SDK CPP V2 v1.33.0
- AWS SDK CPP 1.11.433

Configure the repo with the following command:

```
cd IotMI-DeviceSDK-MatterPlugin
mkdir build
cd build
cmake ..
```

Then build it with the following command:

```
cmake --build .
```

After building, add the library path to LD_LIBRARY_PATH:

```
export LD_LIBRARY_PATH=/path/to/libraries:$LD_LIBRARY_PATH
```

Quickstart: Set Up and Run the Matter Plugin

Follow the steps below to set up the Matter Plugin and the corresponding Matter solution. The flow assumes two machines: a Hub (running the Hub SDK + Matter Plugin) and a Raspberry Pi that represents the "Mobile App" for commissioning and basic control.

Node ID Management

In Matter, every node on a fabric—including devices, controllers, and commissioners—must have a unique Node ID. To avoid collisions, a consistent allocation policy is required. In this guide, Node IDs are assigned manually; in production, you should manage Node IDs properly.

For the examples in this document, the Hub's chip-tool (used by the Matter Plugin) uses default Node ID 112233, and the chip-tool on the Mobile App Raspberry Pi uses Node ID 123456. Newly commissioned devices are assigned non-conflicting Node IDs on the same fabric (for instance, 101 in the quickstart).

Prerequisites

Before you begin, ensure that you have the following:

- Your Hub is already onboarded to Managed integrations and the Hub SDK is deployed (via script or systemd). If not already onboarded, follow Hub onboarding setup to onboard to Managed integrations, and Install and validate the managed integrations Hub SDK to run the hub SDK. Once onboarded, make a note of your hub's managed thing ID. This will be required later.
- You have a Raspberry Pi (or any machine) that can run both chip-tool and AWS CLI.
- You have a Matter device for testing. It can be a real matter device or a virtual Matter device (e.g., lighting-app)

Step 1. Prepare the Raspberry Pi (representing the Mobile App)

- Install AWS CLI and build and install chip-tool
- Build chip-tool following the official chip-tool build guide. The version we've verified is v1.4.2.0.
- Install AWS CLI following the official instructions.
- Prepare persistent storage for chip-tool

```
mkdir -p $HOME/iotmi/matter/
```

- Generate the commissioner certificate chain (assign node ID 123456 to this chip-tool)

```
cd connectedhomeip/
cd out/chip-tool/
./chip-tool pairing get-commissioner-root-certificate \
    --commissioner-nodeid 123456 \
    --storage-directory $HOME/iotmi/matter/
```

The generated chain is stored at: `$HOME/iotmi/matter/chip_tool_config.alpha.ini`

You'll copy this file to the Hub later.

Step 2. Prepare the Hub (which runs the Matter Plugin)

- Build or Install chip-tool on the Hub (the Matter Plugin uses it to perform Matter operations). You can refer to the chip-tool build guide. The version we've verified is v1.4.2.0.

We'll also need the the sha256sum of the chip-tool for later use. You can use the following command to get the sha256sum:

```
sha256sum /path/to/chip-tool
```

- Prepare storage and copy the commissioner file from the Raspberry Pi

```
mkdir -p $HOME/iotmi/matter/
# From Raspberry Pi to Hub (example):
# scp $HOME/iotmi/matter/chip_tool_config.alpha.ini user@HUB_HOST:$HOME/iotmi/matter/
```

- Run the Matter Plugin (provide chip-tool path, its SHA256, and the storage folder)

```
./iotmi_matter_plugin \
    --chip-tool-path /path/to/chip-tool \
    --sha256sum SHA256SUM_OF_THE_CHIP_TOOL \
    --storage-folder $HOME/iotmi/matter/ \
    --node-id 112233
```

The steps below are to be run on the Raspberry Pi i.e. the Commissioner.

Step 3. Commission a Matter device (on the Raspberry Pi)

Use chip-tool to commission the device using its decoded QR code and provision Wi-Fi credentials. In this example, the device will use node ID 101 and the commissioner node ID is 123456.

```
./chip-tool pairing code-wifi 101 \
    YOUR_WIFI_SSID YOUR_WIFI_PW  \
    MT:MFAA0W8C00UFQV2VL00 \
    --bypass-attestation-verifier 1 \
    --commissioner-nodeid 123456 \
    --storage-directory $HOME/iotmi/matter/
```

Notes:

- `101` is the device's Matter node ID (you may choose a different value).
- `MT:MFAA0W8C00UFQV2VL00` is the decoded QR content for the device. To get the decoded context from a QR code, you need to choose a QR code app or library to decode it. You can also use a web service that supports decoding QR code decoding.
- `-bypass-attestation-verifier 1` is for test use only. For production, update the PAA store and perform attestation checks.

chip-tool supports different pairing methods, including QR code or PIN code pairing with Thread or WiFi devices. For more information about the chip-tool pairing command, see the official documentation.

Step 4. Grant Hub access on the device (update ACL)

After commissioning, allow the Hub's chip-tool to control the device. In this example, node ID 123456 (Raspberry Pi) has Admin permission (5) and node ID 112233 (Hub) has Manage permission (4).

```
chip-tool accesscontrol write acl \
    '[{"fabricIndex":1,"privilege":5,"authMode":2,"subjects":[123456], "targets":
 null},{"fabricIndex":1,"privilege":4,"authMode":2,"subjects":[112233], "targets":
 null}]' \
    101 0 \
```

```
    --commissioner-nodeid 123456
    --storage-directory $HOME/iotmi/matter/
```

Step 5. Create a managed thing for the device (User-Guided Setup)

- Start discovery (replace with your Hub's managed thing ID):

```
aws iot-managed-integrations start-device-discovery \
    --discovery-type CUSTOM \
    --custom-protocol-detail '{"Name": "Matter", "NodeId":"101", "FabricId":"1"}' \
    --controller-identifier <HUB_MANAGED_THING_ID>
```

Sample response includes a user-guided setup job ID:

```
{
    "Id": "USER_GUIDED_SETUP_JOB_ID",
    "StartedAt": 1753683326.056
}
```

- Query discovered devices using the job ID:

```
aws iot-managed-integrations \
    list-discovered-devices --identifier <USER_GUIDED_SETUP_JOB_ID>
```

Sample response:

```
{
    "Items": [
        {
            "DeviceTypes": [],
            "DiscoveredAt": "2025-08-05T06:46:35.407000+08:00",
            "AuthenticationMaterial": "<AUTH_MATERIAL>"
        }
    ]
}
```

- Create a managed thing for the device using the AuthenticationMaterial:

```
aws iot-managed-integrations create-managed-thing \
```

```
    --role DEVICE \
    --authentication-material-type DISCOVERED_DEVICE \
    --authentication-material "<AUTH_MATERIAL>"
```

Sample response (with device's managed thing ID):

```
{
    "Id": "DEVICE_MANAGED_THING_ID",
    "Arn": "arn:aws:iotmanagedintegrations:eu-west-1:228183742813:managed-
thing/515cf5a707ec41aaabb9914a1dd2889f",
    "CreatedAt": "2025-08-06T15:00:08.718000+08:00"
}
```

Step 6. Control the device via Managed integrations

Use the `send-managed-thing-command` command to send a command to your managed thing.

```
json=$(jq -cr '.|@json' <<EOF
[
  {
    "endpointId": "1",
    "capabilities": [
      {
        "id": "matter.OnOff@1.4",
        "name": "On/Off",
        "version": "1",
        "actions": [
          {
            "name": "Toggle",
            "parameters": {}
          }
        ]
      }
    ]
  }
]
EOF
)
aws iot-managed-integrations send-managed-thing-command \
    --managed-thing-id "DEVICE_MANAGED_THING_ID" \
    --endpoints "$json"
```

Step 7. Read device state

Send the following command to get the device state.

```
aws iot-managed-integrations get-managed-thing-state \
    --managed-thing-id "DEVICE_MANAGED_THING_ID"
```

Sample result:

```
{
    "Endpoints": [
        {
            "endpointId": "1",
            "capabilities": [
                {
                    "id": "matter.OnOff@1.4",
                    "name": "On/Off",
                    "version": "1.4",
                    "properties": [
                        {
                            "value": {
                                "lastChangedAt": "2025-08-14T13:16:02.132Z",
                                "propertyValue": false
                            },
                            "name": "OnOff"
                        }
                    ]
                }
            ]
        }
    ]
}
```

Step 8. Remove the managed thing from your hub

- Remove the managed thing from your hub with the following command

```
aws iot-managed-integrations delete-managed-thing \
    --identifier "DEVICE_MANAGED_THING_ID"
```

- Unpair the device from the Raspberry Pi's fabric (if desired):

```
./chip-tool pairing unpair 101 \
    --commissioner-nodeid 123456 \
    --storage-directory $HOME/iotmi/matter/
```

Supported Matter device types

Matter device type	Supported Capabilities
OnOffLight	Identify (0x0003), On/Off (0x0006), Level Control (0x0008)
DimmableLight	Identify (0x0003), On/Off (0x0006), Level Control (0x0008)
ColorTemperatureLight	Identify (0x0003), On/Off (0x0006), Level Control (0x0008), Color Control (0x0300)
OnOffPlug-InUint	Identify (0x0003), On/Off (0x0006), Level Control (0x0008)
DimmablePlug-InUnit	Identify (0x0003), On/Off (0x0006), Level Control (0x0008)
OnOffLightSwitch	Identify (0x0003), On/Off (0x0006)
DimmerSwitch	Identify (0x0003), On/Off (0x0006), Level Control (0x0008)
ColorDimmerSwitch	Identify (0x0003), On/Off (0x0006), Level Control (0x0008), Color Control (0x0300)
DoorLock	Identify (0x0003), Door Lock (0x0101)
Thermostat	Identify (0x0003), Thermostat (0x0201)
WaterLeakDetector	Identify (0x0003), Boolean State (0x0045)

Adding Support for Additional Clusters

To extend the Matter Plugin to support additional device types and capabilities, modify **matter_action_converter.cpp** following this pattern:

Implementation pattern:

1. Define mappings for enums and bitmaps
2. Add UpdateState logic for writable attributes
3. Add command processing for cluster actions
4. Add attribute parsing for state reporting

Use existing clusters as templates:

1. **OnOff cluster** - Simplest reference showing all four components with basic enums, writable attributes, commands, and attribute reporting
2. **DoorLock cluster** - Complex example demonstrating multiple enum types, bitmap fields, struct parameters, optional command parameters, and extensive attribute coverage

All supported clusters (Identify, OnOff, LevelControl, DoorLock, Thermostat, ColorControl, BooleanState) follow this structure. Review the existing implementations in matter_action_converter.cpp to understand the complete pattern.

Integrate different Matter Controller solutions

This section provides guidance on how to integrate different Matter Controller solutions with the Matter Plugin. It outlines the possible approaches and important considerations but does not provide detailed implementation code.

There are two main ways to integrate Matter Controller solutions. The first one is using the Matter Plugin with a customized chip-tool. The second one is to use a Matter Controller of your choice and integration with Managed integrations through Custom Protocol library.

Using the Matter Plugin with a Customized Chip-Tool

In this approach, the Matter Plugin is extended to work with a customized version of the Chip-Tool. To achieve this, you may need to modify the Matter Plugin source code and introduce additional logic:

- Adjust STDIO parsing logic: If your customized chip-tool generates logs with a different pattern, update the parsing logic accordingly.
- Implement secure storage: The Matter Plugin provides a file-based storage mechanism for device information. For production use, replace this with a more secure storage implementation or

apply encryption to the data. And it also needs a proper cleanup procedures when the Matter Plugin is uninstalled.

- Implement binary signing and integrity protection: In production deployments, you should enforce integrity protection for both the customized chip-tool and the extended Matter Plugin. This includes signing the binaries as part of your software release process, verifying signatures during startup, and integrating platform security features such as secure boot or OS-level integrity tools.
- Introduce task/event rate limiting: To prevent overload, ensure the Matter Plugin includes proper task and event rate limiting. In production deployments, you should also add basic metrics and monitoring to detect abnormal update patterns and temporarily throttle or suspend the affected device or subscription. This circuit-breaker–like behavior is not provided by the reference implementation and must be implemented by the customer.
- Integrate chip-tool main function logic directly: If you prefer not to rely on STDIO, you can integrate the chip-tool's main function into the Matter Plugin. Standard input/output logic can then be connected to the read/write functions of the ChptoolProc class.
- Support code generation: Implement a mechanism to handle Matter specification version updates through code generation.
- Implement Matter Administration features, including:
  - Assigning Node IDs to the Commissioner, Controller, and Matter devices.
  - Managing multiple Fabrics.

On the chip-tool side, the following enhancements are also recommended:

- Use secure storage: chip-tool stores Matter certificates, private keys, and statistics in local directories. Replace this with a secure implementation.
- Enable parallel processing: By default, Chip-Tool executes one command at a time. Adding parallel execution support may improve efficiency in certain scenarios.

Using an Existing Matter Controller with Managed integrations

If your Matter Controller is not based on chip-tool (for example, a Python-based implementation or function-call–based solution), the STDIO approach may not be suitable. In such cases, you can integrate directly with Managed integrations using a Custom protocol plugin, while using the Matter Plugin as a reference. The following considerations apply:

- Maintain Fabric and Node IDs: Ensure that device metadata, such as adding/removing devices and caching attributes, is consistently maintained.
- Manage subscriptions: Each device should maintain up to one active subscription, so that its state can be updated continuously.
- Propagate state changes: When a device updates its state, verify changes and propagate events to Managed integrations.
- Implement a Matter data model translator: Although Managed integrations uses the Matter data model, its representation is in JSON format. A translator is required to map between your Matter data model format and the JSON representation.

Hub SDK client

The Hub SDK Client library serves as an interface between the managed integrations Hub SDK and your own protocol stack running on the same hub. It exposes a set of public APIs to facilitate your protocol stack's interaction with the Device Hub SDK components. The use cases include custom plugin control, custom plugin provisioner, and local controller.

Topics

- Get your managed integrations Hub SDK
- About the Hub SDK toolkit
- Create your custom application with the Hub SDK client
- Running your custom application
- Hub SDK client API reference
- Data types

Get your managed integrations Hub SDK

The Hub SDK Client comes with the managed integrations SDK. Contact us from the managed integrations console to access the hub SDK.

About the Hub SDK toolkit

After download, you will see a `IotMI-DeviceSDK-Toolkit` folder, which contains all public header files and `.so` files that you can consume in your application. The managed integrations

team also provides an example `main.cpp` for demo purpose, along with the demo application binary under `bin/` that you can directly execute. You can optionally use this as the starting point for your application.

Create your custom application with the Hub SDK client

Use the following steps to create your custom application.

1. Include the header files (.h) and shared object files (.so) in your application.

   You must include the public header files (.h) and shared object files(.so) in your application. For the .so files, you can place them in a lib folder. The final layout will be similar to this:

   ```
   ### include
   #   ### iotmi_device_sdk_client
   #   #   ### iotmi_device_sdk_client_common_types.h
   #   ### iotmi_device_sdk_client.h
   #   ### iotshd_status.h
   ### lib
   #   ### libiotmi_devicesdk_client_module.so
   #   ### libiotmi_log_c.so
   ```

2. Create a Hub SDK client in your main application.

   a. In your main application, you must first initialize the Hub SDK client before it can be used to process requests. You can simply construct the client with a `clientId`.

   b. After you have the client, you can connect it to the managed integrations Device SDK.

      The following is an example of creating the Hub SDK client and how to connect.

      ```
      #include <cstdlib>
      #include <string>
      #include "iotshd_status.h"
      #include "iotmi_device_sdk_client.h"

      auto client = std::make_unique<DeviceSDKClient>(your_own_clientId);
      iotmi_statusCode_t status = client->connect();
      ```

> **Note**
>
> *your_own_clientId* must be the same as the one you specify in start-device-discovery in the user-guided setup, or for create-managed-thing in the simple setup provisioning flow.

3. Publish and subscribe by doing the following steps.

   a. After the connection is established, you can now subscribe to incoming tasks from the managed integrations Hub SDK. The incoming tasks can be control tasks or provision tasks. You also need to define your own callback function upon tasks received, and your own custom context for your own tracing purposes.

```
    // subscribe to provisioning tasks
    iotmi_statusCode_t status = client->iotmi_provision_subscribe_to_tasks(
    example_subscriber_callback, custom_context);

    // subscribe to control tasks
    iotmi_statusCode_t status = client->iotmi_control_subscribe_to_tasks(
    example_subscriber_callback, custom_context);
```

   b. After the connection is established, you can now publish requests from your application to the managed integrations Hub SDK. You can define your own task message type, with different payloads for different business purposes. The requests can include both control requests and provision requests, similar to the subscribe flow. Finally, you can assign an address for your `rspPayload` to get the response from the managed integrations Hub SDK in a synchronized manner.

```
// publish control request
  iotmi_client_request_t api_payload = {
            .messageType = C2MIMessageType::C2MI_CONTROL_EVENT,
            .reqPayload = (uint8_t *)"define_your_req_payload",
            .rspPayload = (uint8_t *)calloc(1000, sizeof(uint8_t))
  };

  status = client->iotmi_control_publish_request(&api_payload);

  // publish provision request
  iotmi_client_request_t api_payload = {
            .messageType = C2MIMessageType::C2MI_DEVICE_ONBOARDED,
```

```
            .reqPayload = (uint8_t *)"define_your_req_payload",
            .rspPayload = (uint8_t *)calloc(1000, sizeof(uint8_t))
};
status = client->iotmi_provision_publish_request(&api_payload);
```

4. Build your own `CMakeLists.txt` and build your application from there. The final output could be a executable binary such as `MyFirstApplication`

Running your custom application

Before you run your custom application, complete the following steps to set up your hub and start the managed integrations Hub SDK:

- Follow the onboarding instructions at Onboard your hubs to Managed Integrations.
- Complete the installation process documented in Install and validate the managed integrations Hub SDK.

Once the prerequisites are met, you can run your custom application. For example:

```
./MyFirstApplication
```

Important

You must manually update the start script listed in Deploy the Hub SDK with a script with a script to start your own application. The order matters, don't change the order.
Update the following. Change

```
./IotMI-DeviceSDK-Toolkit/bin/DeviceSDKClientDemo >> $LOGS_DIR/
logDeviceSDKClientDemo_logs.txt &
```

to

```
./MyFirstApplication >> $LOGS_DIR/MyFirstApplication_logs.txtt &
```

Hub SDK client API reference

The Hub SDK client (DeviceSDKClient class) provides an interface for your custom application to interact with the managed integrations Device SDK. With this client, you can do the following:

- Subscribe to provision-related and control-related tasks from the managed integrations components.
- Publish provision-related and control-related requests to the managed integrations components.

For information about managed integrations for AWS IoT Device Management APIs, see What is AWS Lambda.

Topics

- Client initialization
- Provision task subscription
- Provision task publication
- Control task subscription
- Control task publication
- Logging features
- Other APIs

Client initialization

To start using the `DeviceSDKClient`, initialize it with a client ID.

```
iotmi_statusCode_t DeviceSDKClient(const std::string& clientId)
```

This creates a new `DeviceSDKClient` instance with the specified `clientId`. The `clientId` must match the one you register with managed integrations.

Parameters

`clientId` (string) - The client ID for this instance.

```
connect()
```

Connects the `DeviceSDKClient` instance to managed integrations.

Returns

- `IOTMI_STATUS_OK` - The connection was successful.
- `IOTMI_STATUS_CUSTOM_PLUGIN_CONNECTION_ERROR` - An error occurred while connecting to managed integrations.

Provision task subscription

Use these methods to subscribe to provision-related tasks from the managed integrations components.

```
iotmi_statusCode_t
 iotmi_provision_subscribe_to_tasks(DeviceSDKClient_SubscriberCallback callback, char*
 context)
```

Subscribes to provision-related tasks, such as device onboarding and deprovisioning, from the managed integrations components.

Parameters

- `callback` (DeviceSDKClient_SubscriberCallback) - A callback function that executes when a task is received.
- `context` (char*) - A custom context passed to the callback function.

Returns

- `IOTMI_STATUS_OK` - The subscription was successful.
- `IOTMI_STATUS_CUSTOM_PLUGIN_CLIENT_NOT_CONNECTED` - The DeviceSDKClient instance is not connected to managed integrations.
- `IOTMI_STATUS_CUSTOM_PLUGIN_SUBSCRIBE_ERROR` - An error occurred while subscribing to the tasks.

Provision task publication

Use these methods to publish provision-related requests to the managed integrations components.

```
iotmi_statusCode_t iotmi_provision_publish_request(DataModel::iotmi_client_request_t
 request)
```

Publishes a provision-related request to the managed integrations components. For example, a device onboarded event or deprovisioning status

Parameters

`request` (DataModel::iotmi_client_request_t) - A pointer to a request structure containing the details.

Returns

- `IOTMI_STATUS_OK` - The request was published successfully.
- `IOTMI_STATUS_CUSTOM_PLUGIN_CLIENT_NOT_CONNECTED` - The `DeviceSDKClient` instance is not connected to managed integrations.
- `IOTMI_STATUS_INVALID_PARAMETER` - One or more parameters in the request are invalid.
- `IOTMI_STATUS_INVALID_JSON_OBJECT` - The request payload is not a valid JSON object.
- `IOTMI_STATUS_NO_MEMORY` - A memory allocation error occurred.

Control task subscription

Use these methods to subscribe to control-related tasks from the managed integrations components.

```
iotmi_statusCode_t iotmi_control_subscribe_to_tasks(DeviceSDKClient_SubscriberCallback
 callback, char context)
```

Subscribes to control-related tasks (for example, device control requests) from the managed integrations components.

Parameters

- `callback` (DeviceSDKClient_SubscriberCallback) - A callback function that executes when a task is received.
- `context` (char) - A custom context passed to the callback function.

Returns

- `IOTMI_STATUS_OK` - The subscription was successful.
- `IOTMI_STATUS_CUSTOM_PLUGIN_CLIENT_NOT_CONNECTED` - The `DeviceSDKClient` instance is not connected to managed integrations.

- `IOTMI_STATUS_CUSTOM_PLUGIN_SUBSCRIBE_ERROR` - An error occurred while subscribing to the tasks.

Control task publication

Use these methods to publish control-related requests to the managed integrations components.

```
iotmi_statusCode_t iotmi_control_publish_request(DataModel::iotmi_client_request_t
 request)
```

Publishes a control-related request to the managed integrations components. For example, unsolicited events, command requests, or device state queries.

Parameters

`request` (DataModel::iotmi_client_request_t) - A pointer to a request structure containing the details.

Returns

- `IOTMI_STATUS_OK` - The request was published successfully.
- `IOTMI_STATUS_CUSTOM_PLUGIN_CLIENT_NOT_CONNECTED` - The DeviceSDKClient instance is not connected to managed integrations.
- `IOTMI_STATUS_INVALID_PARAMETER` - One or more parameters in the request are invalid.
- `IOTMI_STATUS_INVALID_JSON_OBJECT` - The request payload is not a valid JSON object.
- `IOTMI_STATUS_NO_MEMORY` - A memory allocation error occurred.

Logging features

Use these methods to implement logging features that managed integrations provides.

Logger initialization

```
void iotmi_devicesdk_log_init(const char* logger_name)
```

You must initialize the logger before you use any logging functionality.

Parameters

`logger_name` - The logger name you specify. Default value is: `MyApplication`

Logging macros

`LOGGER_LOGD(...)`

Use this macro in your application for DEBUG level logging.

`LOGGER_LOGI(...)`

Use this macro in your application for INFO level logging.

`LOGGER_LOGW(...)`

Use this macro in your application for WARN level logging.

`LOGGER_LOGE(...)`

Use this macro in your application for ERROR level logging.

> **Note**
>
> For more information about logging features, see Hub logging documentation. Custom protocol plugins fully support all logging features that managed integrations offers.

Other APIs

```
std::string get_client_id()
```

Returns the client ID associated with the DeviceSDKClient instance.

Returns

The client ID.

Data types

This section defines the data types used for the custom protocol plugin.

iotmi_client_request_t

Represents a request to be published to the managed integrations components.

messageType

The type of the message (CommonTypes::C2MIMessageType). The following list shows the valid values.

- `C2MI_DEVICE_ONBOARDED`: Indicates a device onboarding message with related payload.
- `C2MI_DE_PROVISIONING_PRE_ASSOCIATED_COMPLETE`: Indicates a de-provision task complete notification for a pre-associated device.
- `C2MI_DE_PROVISIONING_ACTIVATED_COMPLETE`: Indicates a de-provision task complete notification for an activated device.
- `C2MI_DE_PROVISIONING_COMPLETE_RESPONSE`: Indicates a de-provision task complete response.
- `C2MI_CONTROL_EVENT`: Indicates a control event with potential device state change.
- `C2MI_CONTROL_SEND_COMMAND`: Indicates a control command from a local controller.
- `C2MI_CONTROL_SEND_DEVICE_STATE_QUERY`: Indicates a control device state query from a local controller.

reqPayload

The request payload, typically a JSON-formatted string.

rspPayload

The response payload, populated by the managed integrations components.

iotmi_client_event_t

Represents an event received from the managed integrations components.

event_id

The unique identifier of the event.

length

The length of the event data.

data

A pointer to the event data, including the `messageType`. The following list shows the possible values.

- `C2MI_PROVISION_UGS_TASK`: Indicates a provision task for the UGS flow.
- `C2MI_PROVISION_SS_TASK`: Indicates a provision task for the SimpleSetup flow.

- `C2MI_DE_PROVISION_PRE_ASSOCIATED_TASK`: Indicates a de-provision task for a pre-associated device.
- `C2MI_DE_PROVISION_ACTIVATED_TASK`: Indicates a de-provision task for an activated device.
- `C2MI_DEVICE_ONBOARDED_RESPONSE`: Indicates a device onboarding response.
- `C2MI_CONTROL_TASK`: Indicates a control task.
- `C2MI_CONTROL_EVENT_NOTIFICATION`: Indicates a control event notification for a local controller.

ctx

A custom context associated with the event.

Hub control

Hub control is an extension to the managed integrations End device SDK that allows it to interface with the `MQTTProxy` component in the Hub SDK. With hub control, you can implement code using the End device SDK and control your hub through the Managed Integrations cloud as a separate device. The hub control SDK will be provided as a separate package with-in the Hub SDK, labeled as `iot-managed-integrations-hub-control-x.x.x`.

Topics

- Prerequisites
- End device SDK components
- Integrate with the End device SDK
- Example: Build hub control
- Supported examples
- Supported platforms

Prerequisites

To set up hub control, you need the following:

- A hub onboarded to the Hub SDK, version 0.4.0 or greater.
- Download the latest version of the End device SDK from the AWS Management Console.

- An MQTT proxy component running on the hub, version 0.5.0 or greater.

End device SDK components

Use the following components from the End device SDK:

- Code generator for the data model
- Data model handler

Since the Hub SDK already has an onboarding process and a connection to the cloud, you don't need the following components:

- Provisionee
- PKCS interface
- Jobs handler
- MQTT Agent

Integrate with the End device SDK

1. Follow the instructions in Code generator for Data Model to generate the low level C code.
2. Follow the instructions in Integrating the End device SDK to:

    a. **Set up the build environment**

    Build the code on Amazon Linux 2023/x86_64 as your development host. Install the necessary build dependencies:

    ```
    dnf install make gcc gcc-c++ cmake
    ```

    b. **Develop hardware callback functions**

    Before implementing the hardware callback functions, understand how the API works. This example uses the On/Off cluster and OnOff attribute to control a device function. For API details, see Low level C-Function APIs.

    ```
    struct DeviceState
    {
    ```

```
  struct iotmiDev_Agent *agent;
  struct iotmiDev_Endpoint *endpointLight;
  /* This simulates the HW state of OnOff */
  bool hwState;
};

/* This implementation for OnOff getter just reads
   the state from the DeviceState */
iotmiDev_DMStatus exampleGetOnOff(bool *value, void *user)
{
   struct DeviceState *state = (struct DeviceState *)(user);
  *value = state->hwState;
  return iotmiDev_DMStatusOk;
}
```

c. **Set up endpoints and hook hardware callback functions**

After implementing the functions, create endpoints and register your callbacks. Complete these tasks:

i. Create a device agent

ii. Fill callback function points for each cluster struct you want to support

iii. Set up endpoints and register supported clusters

```
struct DeviceState
{
    struct iotmiDev_Agent * agent;
    struct iotmiDev_Endpoint *endpoint1;

    /* OnOff cluster states*/
    bool hwState;
};


/* This implementation for OnOff getter just reads
   the state from the DeviceState */
iotmiDev_DMStatus exampleGetOnOff( bool * value, void * user )
{
    struct DeviceState * state = ( struct DeviceState * ) ( user );
    *value = state->hwState;
    printf( "%s(): state->hwState: %d\n", __func__, state->hwState );
    return iotmiDev_DMStatusOk;
```

```
}

iotmiDev_DMStatus exampleGetOnTime( uint16_t * value, void * user )
{
    *value = 0;
    printf( "%s(): OnTime is %u\n", __func__, *value );
    return iotmiDev_DMStatusOk;
}

iotmiDev_DMStatus exampleGetStartUpOnOff( iotmiDev_OnOff_StartUpOnOffEnum *
 value, void * user )
{
    *value = iotmiDev_OnOff_StartUpOnOffEnum_Off;
    printf( "%s(): StartUpOnOff is %d\n", __func__, *value );
    return iotmiDev_DMStatusOk;
}

void setupOnOff( struct DeviceState *state )
{
    struct iotmiDev_clusterOnOff clusterOnOff = {
        .getOnOff = exampleGetOnOff,
        .getOnTime = exampleGetOnTime,
        .getStartUpOnOff = exampleGetStartUpOnOff,
    };
    iotmiDev_OnOffRegisterCluster( state->endpoint1,
                                   &clusterOnOff,
                                   ( void * ) state);
}


/* Here is the sample setting up an endpoint 1 with OnOff
   cluster. Note all error handling code is omitted. */
void setupAgent(struct DeviceState *state)
{
    struct iotmiDev_Agent_Config config = {
        .thingId = IOTMI_DEVICE_MANAGED_THING_ID,
        .clientId = IOTMI_DEVICE_CLIENT_ID,
    };
    iotmiDev_Agent_InitDefaultConfig(&config);

    /* Create a device agent before calling other SDK APIs */
    state->agent = iotmiDev_Agent_new(&config);

    /* Create endpoint#1 */
```

```
    state->endpoint1 = iotmiDev_Agent_addEndpoint( state->agent,
                                                   1,
                                                   "Data Model Handler Test
 Device",
                                                   (const char*[])
{ "Camera" },
                                                   1 );
    setupOnOff(state);
}
```

Example: Build hub control

Hub control is provided as part of the Hub SDK package. The hub control sub-package is labeled with `iot-managed-integrations-hub-control-x.x.x` and contains different libraries than the unmodified device SDK.

1. Move the code generated files to the `example` folder:

```
cp codegen/out/* example/dm
```

2. To build hub control, run the following commands:

```
cd <hub-control-root-folder>
```

```
mkdir build
```

```
cd build
```

```
cmake -DBUILD_EXAMPLE_WITH_MQTT_PROXY=ON -
DIOTMI_USE_MANAGED_INTEGRATIONS_DEVICE_LOG=ON ..
```

```
cmake -build .
```

3. Run the examples with the `MQTTProxy` component on the hub, with the `HubOnboarding` and `MQTTProxy` components running.

```
./examples/iotmi_device_sample_camera/iotmi_device_sample_camera
```

See Managed Integrations data model for the data model. Follow Step 5 in Get started with End device SDK to set up endpoints and manage communications between the end-user and iot-managed-integrations.

Supported examples

The following examples have been built and tested:

- iotmi_device_dm_air_purifier_demo
- iotmi_device_basic_diagnostics
- iotmi_device_dm_camera_demo

Supported platforms

The following table displays the supported platforms for hub control.

Architecture	Operating system	GCC version	Binutils version
X86_64	Linux	10.5.0	2.37
aarch64	Linux	10.5.0	2.37

Enable CloudWatch Logs

The Hub SDK provides comprehensive logging functionality. By default, the Hub SDK writes logs to the local file system. However, you can leverage the cloud API to configure log streaming to CloudWatch Logs, which offers:

- **Monitor device performance**: Capture detailed runtime logs for proactive device management. Enable advanced log analysis and monitoring across your device fleet
- **Troubleshoot issues**: Generate granular log entries for rapid diagnostic analysis. Record system and application-level events for in-depth investigation.
- **Flexible and centralized logging**: Remote log management without direct device access. Aggregate logs from multiple devices in a single, searchable repository.

Prerequisites

- Onboard the managed device to the cloud. See Hub onboarding setup for details.
- Verify Hub agent startup and successful initialization. See Install and validate the managed integrations Hub SDK for details.

> **Note**
>
> To create logging configurations, see PutRuntimeLogConfiguration API for details.

> **Warning**
>
> Enabling logs counts towards the tiered quota metering. Increasing log levels will result in higher message volume and additional costs.

Setup Hub SDK log configurations

Configure the hub SDK log settings by calling the API to set up runtime log configuration.

Example sample API Request

```
aws iot-managed-integrations put-runtime-log-configuration \
    --managed-thing-id MANAGED_THING_ID \
    --runtime-log-configurations LogLevel=DEBUG,UploadLog=TRUE
```

RuntimeLogConfigurations attributes

The following attributes are optional and can be configured in the `RuntimeLogConfigurations` API.

LogLevel

Sets minimum severity level for runtime traces. Values: `DEBUG, ERROR, INFO, WARN`

Default: `WARN` (released build)

LogFlushLevel

Determines severity level for immediate data flushing to local storage. Values: `DEBUG, ERROR, INFO, WARN`

Default: `DISABLED`

LocalStoreLocation

Specifies storage location for runtime traces. Default: `/var/log/awsiotmi`

- Active log: `/var/log/awsiotmi/ManagedIntegrationsDeviceSdkHub.log`
- Rotated logs: `/var/log/awsiotmi/ManagedIntegrationsDeviceSdkHub.N.log` (N indicates rotation order)

LocalStoreFileRotationMaxBytes

Triggers file rotation when current file exceeds specified size.

> **Important**
>
> For optimal efficiency, keep file size below 125 KB. Values above 125 KB will be automatically limited.

LocalStoreFileRotationMaxFiles,

Sets maximum number of rotation files allowed by log daemon.

UploadLog

Controls runtime trace transfer to cloud. Logs are stored in `/aws/iotmanagedintegration` CloudWatch Logs group.

Default: `false`.

UploadPeriodMinutes

Defines frequency of runtime trace uploads. Default: 5

DeleteLocalStoreAfterUpload

Controls file deletion after upload. Default: `true`

> **Note**
>
> If set to false, uploaded files are renamed to: `/var/log/awsiotmi/ManagedIntegrationsDeviceSdkHub.uploaded.{uploaded_timestamp}`

Example log file

See example of a CloudWatch Logs file below:

```
2024-11-29T09:00:56.713Z {"messageId":"aa11d8c62aa24fd685b1a9b1f4090eab","details":"[23986616][JOBS_HANDLER][ERROR] Failed to get Job ID …

{
    "messageId": "aa11d8c62aa24fd685b1a9b1f4090eab",
    "details": "[23986616][JOBS_HANDLER][ERROR] Failed to get Job ID from the message.\r\n[23986616][JOBS_HANDLER][INFO] Sent describe next job.\r\n[2398661
to get Job ID from the message.\r\n\r\n[24017056][JOBS_HANDLER][INFO] Sent Describe Next.\r\n",
    "resourceId": "1fdf4a6a4ecb4cfdbdec1e736ca46bf5"
}
2024-11-29T09:01:57.157Z {"messageId":"63f7a48efa94417694e36f45c23ffa86","details":"[24047493][JOBS_HANDLER][ERROR] Failed to get Job ID …

{
    "messageId": "63f7a48efa94417694e36f45c23ffa86",
    "details": "[24047493][JOBS_HANDLER][ERROR] Failed to get Job ID from the message.\r\n[24047493][JOBS_HANDLER][INFO] Sent describe next job.\r\n[2404749
to get Job ID from the message.\r\n[24077937][JOBS_HANDLER][INFO] Sent Describe Next.\r\n\r\n",
    "resourceId": "1fdf4a6a4ecb4cfdbdec1e736ca46bf5"
}
2024-11-29T09:02:57.592Z {"messageId":"dfa41ca99a53440584df29dd892a1a23","details":"[24108373][JOBS_HANDLER][ERROR] Failed to get Job ID …

{
    "messageId": "dfa41ca99a53440584df29dd892a1a23",
    "details": "[24108373][JOBS_HANDLER][ERROR] Failed to get Job ID from the message.\r\n[24108373][JOBS_HANDLER][INFO] Sent describe next job.\r\n[2410837
message.Sent describe next job.\r\n\r\n[24138811][JOBS_HANDLER][INFO] Sent Describe Next.\r\n",
```

Supported Zigbee and Z-Wave device types

This page lists the hub-connected device types that have been tested with managed integrations and are supported. Managed Integrations supports both Simple setup (SS) and User guided setup (UGS) for these devices.

This table lists the supported Zigbee devices.

Zigbee device type	Supported capabilities
Smart bulb / Dimmable light / RGB light	OnOff, LevelControl, ColorControl
Smart plug	OnOff
Smart switch	OnOff

Zigbee device type	Supported capabilities
LED strip	OnOff, LevelControl, ColorControl
Water valve	OnOff
Radiator valve	Thermostat, OnOff, Timer
Thermostat	Thermostat, FanControl, OnOff, Timer
Garage door opener	WindowCovering, OnOff, LevelControl
Smoke alarm	BooleanState, OnOff, TemperatureMeasurement, Timer, SmokeCOAlarm
Motion sensor	BooleanState
Occupancy/Human presence sensor	BooleanState, OccupancySensing
Door and window sensor	BooleanState
Water leak sensor	BooleanState
Vibration sensor	BooleanState
Temperature and humidity sensor	TemperatureMeasurement, RelativeHumidityMeasurement

This table lists the supported Z-Wave devices.

Z-Wave device type	Supported capabilities
Smart bulb / Dimmable light	OnOff, LevelControl
Smart plug	OnOff
Garage door controller	OnOff, LevelControl
Energy meter	ElectricalEnergyMeasurement, ElectricalPowerMeasurement

Z-Wave device type	Supported capabilities
Battery	LevelControl
Siren	LevelControl
Motion sensor	BooleanState
Door and window sensor	BooleanState
Water leak sensor	BooleanState
Temperature sensor	TemperatureMeasurement
CO sensor	SmokeCOAlarm
Smoke sensor	SmokeCOAlarm

Run managed integrations on Raspberry Pi

Note

This implementation of AWS IoT Hub SDK on Raspberry Pi is a demonstration project intended for learning and testing purposes only and is not intended to be used in production environments. For the purposes of this demo, set the following configurations for development ease:

AWS credentials storage: For demo purposes only, credentials and certificates are stored in an accessible location for easier testing and development. Production environments must use secure storage solutions like AWS Secrets Manager, or Systems Manager Parameter Store. They must implement encryption at rest, and follow AWS IoT security guidelines.

Container privileges: The demo runs with elevated privileges to allow unrestricted access to host resources and simplify development workflows. In production, containers should operate with minimal required privileges.

Network bridge configuration: The demo uses a network bridge configuration that exposes internal network traffic for easier debugging and monitoring. In production environments, implement proper network isolation and segmentation to prevent unauthorized access to internal network traffic.

USB device permissions: Unrestricted USB device access is enabled to facilitate easy connection of development peripherals and testing devices. For production, implement strict USB device controls and validation to prevent device spoofing attacks.
These configurations enable straightforward testing and must notcbe used in production environments. When deploying to production, please follow security best practices to prevent host system compromise and unauthorized access to credentials.

As a prerequisite, you must set up the Sonoff Zigbee USB dongle before setting up the Raspberry Pi.

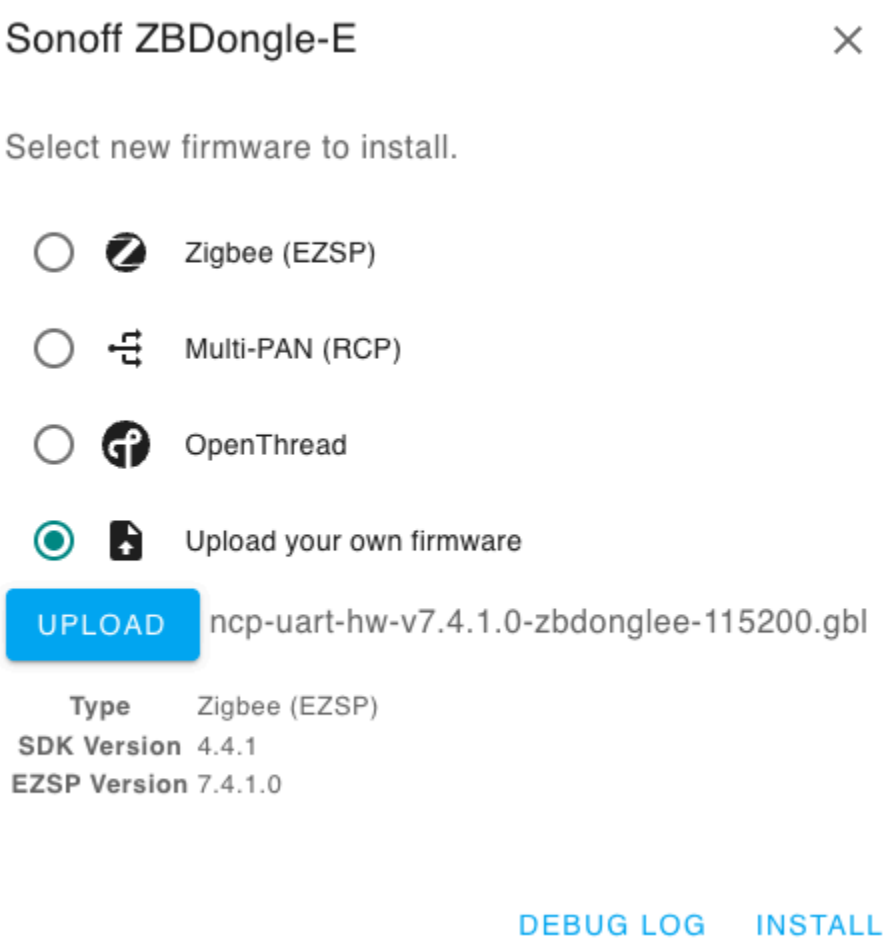
Flash firmware to Sonoff Zigbee USB dongle

Prerequisites

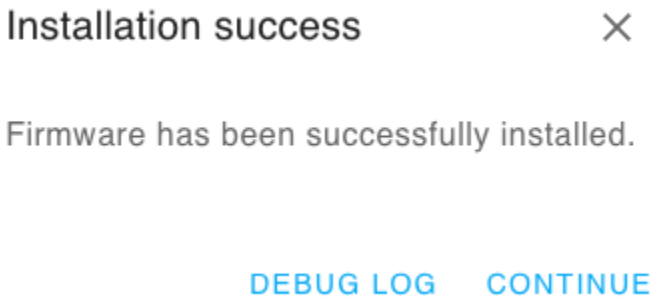
- Sonoff Zigbee USB Dongle
- Windows: Install CP210x Universal Windows Driver

Flash the firmware

1. Download Zigbee Dongle Firmware Build 7.4.1.0.
2. Open Silabs Firmware Flasher.
3. Connect the Sonoff Zigbee USB Dongle to your computer.
4. Scroll and find **ZBDongle-E**.
5. Choose **Connect**.
6. Wait for the device to connect.
7. Choose **Change firmware**.
8. Select **Upload your own firmware**.
9. Find the location of Zigbee Dongle Firmware Build 7.4.1.0 download and select it.



10. Click **Install**.
11. Wait for firmware to install.



12. Choose **Continue** when installation is complete.



The dongle is now ready for use.

Choose between the below listed options to run the managed integrations Hub SDK on your Raspberry Pi. The setup and validation steps for both approaches are listed below.

Topics

- Managed Integrations Hub SDK image on Raspberry Pi
- Managed Integrations Hub SDK Docker container on Raspberry Pi
- Managed Integrations demo application

Managed Integrations Hub SDK image on Raspberry Pi

Note

This implementation of AWS IoT Hub SDK on Raspberry Pi is a demonstration project intended for learning and testing purposes only and is not intended to be used in production environments. For the purposes of this demo, set the following configurations for development ease:
AWS credentials storage: For demo purposes only, credentials and certificates are stored in an accessible location for easier testing and development. Production environments must use secure storage solutions like AWS Secrets Manager, or Systems Manager Parameter Store. They must implement encryption at rest, and follow AWS IoT security guidelines.
Container privileges: The demo runs with elevated privileges to allow unrestricted access to host resources and simplify development workflows. In production, containers should operate with minimal required privileges.

Network bridge configuration: The demo uses a network bridge configuration that exposes internal network traffic for easier debugging and monitoring. In production environments, implement proper network isolation and segmentation to prevent unauthorized access to internal network traffic.
USB device permissions: Unrestricted USB device access is enabled to facilitate easy connection of development peripherals and testing devices. For production, implement strict USB device controls and validation to prevent device spoofing attacks.
These configurations enable straightforward testing and must notcbe used in production environments. When deploying to production, please follow security best practices to prevent host system compromise and unauthorized access to credentials.

Prerequisites

Complete these requirements before deploying the Raspberry Pi image:

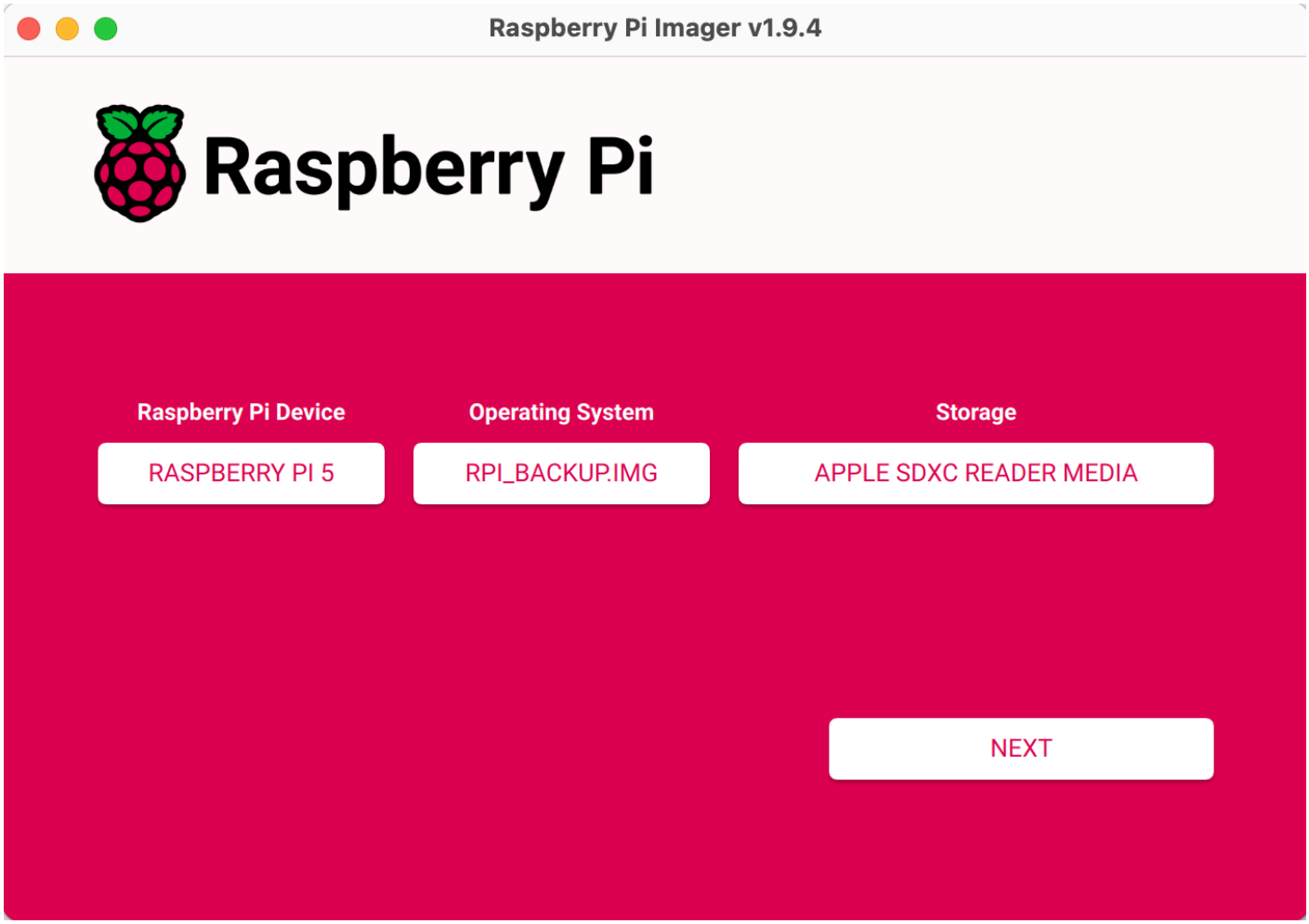
- Download and install Raspberry Pi imager.
- Obtain an SD Card.
- Set up a Raspberry Pi 5 with 2.4Ghz 64-bit quad-core CPU (8GB RAM).
- Connect a Sonoff Zigbee USB Dongle.
- Flash firmware to Sonoff Zigbee USB dongle.
- Connect a Silicon Labs SLUSB001A Dongle.
- Sign up for an AWS account.
- Install the latest version of AWS CLI from the Managed Integrations AWS CLI Command Reference.

Flash a Raspberry Pi image on a new SD card

Flash the Managed Integrations image to your SD card using these steps:

1. Download the Managed Integrations Raspberry Pi Hub SDK Image.
2. Launch Raspberry Pi Imager on your desktop.
3. Insert SD card into your computer's built-in SD card reader, or external USB card reader.
4. Select **Choose Device** → Raspberry Pi 5.
5. Select **Choose OS** → Use custom → Find the IotMI-HubSDK-RPi-Image-v1.0.0.img.gz file → Open.

6. Select **Choose storage** → Select your SD Card Reader.
7. Verify your configuration matches the following screen:



8. Click **Next**.
9. Configure the OS customization settings:
   - **Hostname:** Select **raspberrypi**.
   - **Username and Password:**
     - Enable **Set username and password**:
     - For **Username:**, input hub123456.
     - For **Password:**, input sh123456.
   - **Wireless LAN:**
     - Enable **Configure wireless LAN**.
     - Enter your router's **SSID**, and **password**.

     Example settings:

    - **SSID**:`iotmi-tplink`
    - **Password**:`********` (Minimum of 8 characters)
  - Set **Country:** to US.
- **Set Locale Settings:**
  - Set **Time zone:** to `America/Los Angeles`.
  - Set **Keyboard Layout:** to US.
- **SSH:**
  - Choose the **services** tab.
  - Check **Enable SSH**.
  - Choose **Use password authentication**.

10. Confirm all popups for OS customization and data erasure.
11. Wait for the writing process to complete.
12. Verify successful completion with the following screen:



13. Click **Continue**.
14. Remove the SD card and insert it into your Raspberry Pi.

Run the Hub SDK on the Raspberry Pi

Start the Hub SDK services on your configured Raspberry Pi:

1. Insert the prepared SD card into the Raspberry Pi 5 device.
2. Connect the Sonoff Zigbee USB Dongle and Silicon Labs SLUSB001A Dongle to the Raspberry Pi.
3. Power on the Raspberry Pi.
4. Ensure the Raspberry Pi and your computer (from which you SSH) are on the same network.
5. SSH into the Raspberry Pi using the credentials you set during image deployment.

```
ssh username@hostname
```

6. Navigate to the hub SDK directory:

```
cd /data/aws/iotmi
```

7. Complete the Hub onboarding setup to add authentication and configuration materials.

> **Note**
>
> You must be on YUL or DUB region to do this step.

8. Run the Hub SDK:

```
cd /data/aws/iotmi
bash start_hub_sdk.sh
```

The system displays the following response for a successful Hub SDK start:

```
-------Stopping SDK running processes---
-------Starting Hub SDK------------
-------Creating logs directory----------
Logs directory created.
-------Verifying Middleware paths-------
All middleware libraries exist
```

```
-------Verifying Middleware pre reqs---
AIPC and KVstroage directories exist
-------Starting HubOnboarding-----------
-------Starting MQTT Proxy-----------
-------Staring Log Daemon---
-------Starting Event Manager-----------
-------Starting Zigbee Service----------
--Checking Zigbee network information--
-------Starting Zwave Service----------
/data/aws/iotmi/middleware/AceZwave/bin /data/aws/iotmi
/data/aws/iotmi
-------Starting CDMB--------------------
-------Starting Agent--------------------
-------Starting Provisioner--------------
-------Checking SDK status---------------
hub1234+    1780  0.2  0.1 1093936 16368 pts/1   Sl+  16:34   0:00 ./iotmi_mqtt_proxy -
C /data/aws/iotmi/config/iotmi_config.json
Process 'iotmi_mqtt_proxy' is running.
hub1234+    1884  0.0  0.0 236272  2624 pts/1    Sl+  16:34   0:00 ./middleware/
AceCommon/bin/ace_eventmgr
Process 'ace_eventmgr' is running.
hub1234+    1892  9.1  0.1 393040  8352 pts/1    Sl+  16:34   0:04 ./middleware/
AceZigbee/bin/ace_zigbee_service
Process 'ace_zigbee_service' is running.
hub1234+    1923  0.0  0.1 1570736 12736 pts/1   Sl+  16:34   0:00 ./zwave_svc
Process 'zwave_svc' is running.
hub1234+    1958  0.0  0.0 1067632 5776 pts/1    Sl+  16:34   0:00 ./iotmi_cdmb
Process 'iotmi_cdmb' is running.
hub1234+    2001  0.2  0.2 2017712 21264 pts/1   Sl+  16:35   0:00 ./iotmi_device_agent
Process 'iotmi_device_agent' is running.
hub1234+    2045  0.0  0.1 1457824 12624 pts/1   Sl+  16:35   0:00 ./
iotmi_lpw_provisioner
Process 'iotmi_lpw_provisioner' is running.
hub1234+    1813  0.0  0.0 875152  6848 pts/1    Sl+  16:34   0:00 ./iotmi_log_daemon
Process 'iotmi_log_daemon' is running.
------Successfully Started Hub SDK----
```

Next steps

After successfully starting the Hub SDK, proceed with device onboarding and management at User guided setup to onboard and operate devices.

Managed Integrations Hub SDK Docker container on Raspberry Pi

Note

This implementation of AWS IoT Hub SDK on Raspberry Pi is a demonstration project intended for learning and testing purposes only and is not intended to be used in production environments. For the purposes of this demo, set the following configurations for development ease:
AWS credentials storage: For demo purposes only, credentials and certificates are stored in an accessible location for easier testing and development. Production environments must use secure storage solutions like AWS Secrets Manager, or Systems Manager Parameter Store. They must implement encryption at rest, and follow AWS IoT security guidelines.
Container privileges: The demo runs with elevated privileges to allow unrestricted access to host resources and simplify development workflows. In production, containers should operate with minimal required privileges.
Network bridge configuration: The demo uses a network bridge configuration that exposes internal network traffic for easier debugging and monitoring. In production environments, implement proper network isolation and segmentation to prevent unauthorized access to internal network traffic.
USB device permissions: Unrestricted USB device access is enabled to facilitate easy connection of development peripherals and testing devices. For production, implement strict USB device controls and validation to prevent device spoofing attacks.
These configurations enable straightforward testing and must notcbe used in production environments. When deploying to production, please follow security best practices to prevent host system compromise and unauthorized access to credentials.

Prerequisites

The following prerequisites are required to for the docker container.

- Download and install Raspberry Pi imager.
- Obtain an SD Card.
- Set up a Raspberry Pi 5 with 2.4Ghz 64-bit quad-core CPU (8GB RAM).
- Connect a Sonoff Zigbee USB Dongle.
- Flash firmware to Sonoff Zigbee USB dongle.

- Connect a Silicon Labs SLUSB001A Dongle.
- Sign up for an AWS account.
- Install the latest version of AWS CLI from the Managed Integrations AWS CLI Command Reference.
- SSH access to the Raspberry Pi with IP address or hostname.

Use Managed Integrations Hub SDK Docker container on Raspberry Pi

1. Download Managed Integrations Raspberry Pi Hub SDK Docker.
2. Copy the file to the Raspberry Pi using SCP:

```
scp ~/path/to/IotMI-HubSDK-Docker-v1.0.0.tar.gz [username]@raspberrypi.local:~
```

3. Connect to the Raspberry Pi via SSH:

```
ssh hub123456@raspberrypi.local
```

4. Install Docker if not present:

```
# Install Docker
cd
curl -fsSL https://get.docker.com | sudo sh

# Add your user to docker group
sudo usermod -aG docker $USER
exit # exit ssh

# Log in again
```

5. Install Docker Compose if not present:

```
# Install Docker Compose
sudo apt-get update
sudo apt-get install -y docker-compose-plugin
```

6. Extract the Hub SDK files:

```
# Navigate to the home directory
cd
```

```
# Extract the hub-docker.tar.gz file
tar -xzf IotMI-HubSDK-Docker-v1.0.0.tar.gz
```

7. Navigate to the hub-docker directory:

```
cd IotMI-HubSDK-Docker
```

8. Complete Hub onboarding setup to configure authentication and settings.

> **Note**
>
> You must be on YUL or DUB region to do this step.

9. Start the Docker container:

```
# The first time it's called, it will build the container
docker compose up -d
docker compose logs -f
```

Expected output:

```
[+] Running 1/1
 # Container iotmi-hubsdk-docker-hubsdk-1  Started
hubsdk-1  | -\-\-\-\-\-\-Checking USB dongles-\-\-\-\-\-\-\-\-\-\-\-\-\-
hubsdk-1  | -\-\-\-\-\-\-Stopping SDK running processes-\-\-
hubsdk-1  | iotmi_mqtt_proxy: no process found
hubsdk-1  | ace_eventmgr: no process found
hubsdk-1  | ace_zigbee_service: no process found
hubsdk-1  | zwave_svc: no process found
hubsdk-1  | iotmi_cdmb: no process found
hubsdk-1  | iotmi_device_agent: no process found
hubsdk-1  | iotmi_lpw_provisioner: no process found
hubsdk-1  | iotmi_log_daemon: no process found
hubsdk-1  | -\-\-\-\-\-\-Starting Hub SDK-\-\-\-\-\-\-\-\-\-\-\-\-\-
hubsdk-1  | -\-\-\-\-\-\-Creating logs directory-\-\-\-\-\-\-\-\-\-
hubsdk-1  | Logs directory created.
hubsdk-1  | -\-\-\-\-\-\-Verifying Middleware paths-\-\-\-\-\-\-
hubsdk-1  | All middleware libraries exist
hubsdk-1  | -\-\-\-\-\-\-Verifying Middleware pre reqs-\-\-
hubsdk-1  | AIPC and KVstroage directories exist
hubsdk-1  | -\-\-\-\-\-\-Starting HubOnboarding-\-\-\-\-\-\-\-\-\-\-\-
```

```
hubsdk-1  | -\-\-\-\-\-\-Starting MQTT Proxy-\-\-\-\-\-\-\-\-\-\-
hubsdk-1  | -\-\-\-\-\-\-Staring Log Daemon-\-\-
hubsdk-1  | -\-\-\-\-\-\-Starting Event Manager-\-\-\-\-\-\-\-\-\-\-
hubsdk-1  | -\-\-\-\-\-\-Starting Zigbee Service-\-\-\-\-\-\-\-\-\-\-
hubsdk-1  | -\-\-\-\-\-\-Starting Zwave Service-\-\-\-\-\-\-\-\-\-\-
hubsdk-1  | /data/aws/iotmi/middleware/AceZwave/bin /data/aws/iotmi
hubsdk-1  | /data/aws/iotmi
hubsdk-1  | -\-\-\-\-\-\-Starting CDMB-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-
hubsdk-1  | -\-\-\-\-\-\-Starting Agent-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-
hubsdk-1  | -\-\-\-\-\-\-Starting Provisioner-\-\-\-\-\-\-\-\-\-\-\-\-\-\-
hubsdk-1  | -\-\-\-\-\-\-Checking SDK status-\-\-\-\-\-\-\-\-\-\-\-\-\-\-\-
hubsdk-1  | root         105  0.2  0.2 1093488 16608 ?       Sl   20:51   0:00 ./
iotmi_mqtt_proxy -C /data/aws/iotmi/config/iotmi_config.json
hubsdk-1  | Process 'iotmi_mqtt_proxy' is running.
hubsdk-1  | root         183  0.0  0.0 236272  3152 ?        Sl   20:51   0:00 ./
middleware/AceCommon/bin/ace_eventmgr
hubsdk-1  | Process 'ace_eventmgr' is running.
hubsdk-1  | root         190 12.0  0.1 319264  8352 ?        Sl   20:51   0:04 ./
middleware/AceZigbee/bin/ace_zigbee_service
hubsdk-1  | Process 'ace_zigbee_service' is running.
hubsdk-1  | root         200  0.0  0.1 1365792 12480 ?       Sl   20:51   0:00 ./
zwave_svc
hubsdk-1  | Process 'zwave_svc' is running.
hubsdk-1  | root         233  0.0  0.0 1198704 5760 ?        Sl   20:51   0:00 ./
iotmi_cdmb
hubsdk-1  | Process 'iotmi_cdmb' is running.
hubsdk-1  | root         268  0.2  0.2 2017424 21968 ?       Sl   20:51   0:00 ./
iotmi_device_agent
hubsdk-1  | Process 'iotmi_device_agent' is running.
hubsdk-1  | root         311  0.1  0.1 1523072 13008 ?       Sl   20:51   0:00 ./
iotmi_lpw_provisioner
hubsdk-1  | Process 'iotmi_lpw_provisioner' is running.
hubsdk-1  | root         132  0.0  0.0 875024  7232 ?        Sl   20:51   0:00 ./
iotmi_log_daemon
hubsdk-1  | Process 'iotmi_log_daemon' is running.
hubsdk-1  | -\-\-\-\-\-\-Successfully Started Hub SDK-\-\-\-
```

After successfully starting the Hub SDK, proceed with device onboarding and management at User guided setup to onboard and operate devices.

Note

- To access the Docker container bash shell, run the following command:

```
docker compose exec hubsdk bash
```

- To restart the container after reboot, run the following command:

```
docker compose up -d
```

- To update the Hub SDK, replace binaries in the following folder:

```
hub-docker/iotmi
```

- To safely restart the container while preserving data, do:

```
docker compose down
docker compose up -d
docker compose logs -f
```

Managed Integrations demo application

Note

This implementation of AWS IoT Hub SDK on Raspberry Pi is a demonstration project intended for learning and testing purposes only and is not intended to be used in production environments. For the purposes of this demo, set the following configurations for development ease:
AWS credentials storage: For demo purposes only, credentials and certificates are stored in an accessible location for easier testing and development. Production environments must use secure storage solutions like AWS Secrets Manager, or Systems Manager Parameter Store. They must implement encryption at rest, and follow AWS IoT security guidelines.
Container privileges: The demo runs with elevated privileges to allow unrestricted access to host resources and simplify development workflows. In production, containers should operate with minimal required privileges.
Network bridge configuration: The demo uses a network bridge configuration that exposes internal network traffic for easier debugging and monitoring. In production environments, implement proper network isolation and segmentation to prevent unauthorized access to internal network traffic.

USB device permissions: Unrestricted USB device access is enabled to facilitate easy connection of development peripherals and testing devices. For production, implement strict USB device controls and validation to prevent device spoofing attacks.
These configurations enable straightforward testing and must notcbe used in production environments. When deploying to production, please follow security best practices to prevent host system compromise and unauthorized access to credentials.

The demo application is a React-based demo application showcasing Managed Integrations capabilities for smart home device management. This application demonstrates device onboarding, control, and monitoring for Z-Wave and Zigbee devices through a modern web interface.

Prerequisites

- Sign up for an AWS account.
- Create a credential locker and add the credential locker to your hub .
- Complete Hub onboarding setup.
- Node.js 18+ and npm.
- Install the latest version of AWS CLI from the Managed Integrations AWS CLI Command Reference.
- Modern web browser (Chrome, Firefox, Safari, Edge)

Install and configure the Application

1. Download Managed Integrations demo application.
2. Extract the package:

```
cd ~/Downloads
tar -xzf IotMI-HubSDK-DemoApp-v1.0.0.tar.gz
cd IotManagedIntegrations-DemoApp
```

3. Install dependencies:

```
npm install
```

4. Create a `.env` file in the root directory:

```
# AWS Configuration
REACT_APP_AWS_REGION=your_region
REACT_APP_AWS_ACCESS_KEY_ID=your_access_key
REACT_APP_AWS_SECRET_ACCESS_KEY=your_secret_key
REACT_APP_AWS_SESSION_TOKEN=your_session_token

# IoT Managed Integrations Endpoint
REACT_APP_IOT_ENDPOINT=https://your-iot-endpoint.amazonaws.com

# Hub Configuration
REACT_APP_HUB_MANAGED_THING_ID=your_hub_id
REACT_APP_CREDENTIAL_LOCKER_ID=your_credential_locker_id
```

5. Build and start the application:

```
npm start
```

6. Access the application at:

```
http://localhost:3000
```

For pricing information, refer to Managed Integrations section on the AWS IoT Device Management pricing page.

Offboard managed integrations hub

Overview of Hub SDK offboard process

The hub offboarding process removes a hub from AWS Cloud management system. When the cloud sends a DeleteManagedThing request, the process accomplishes two primary objectives:

Device-side actions:

- Reset the hub's internal state
- Delete all locally saved data
- Prepare the device for potential future re-onboarding

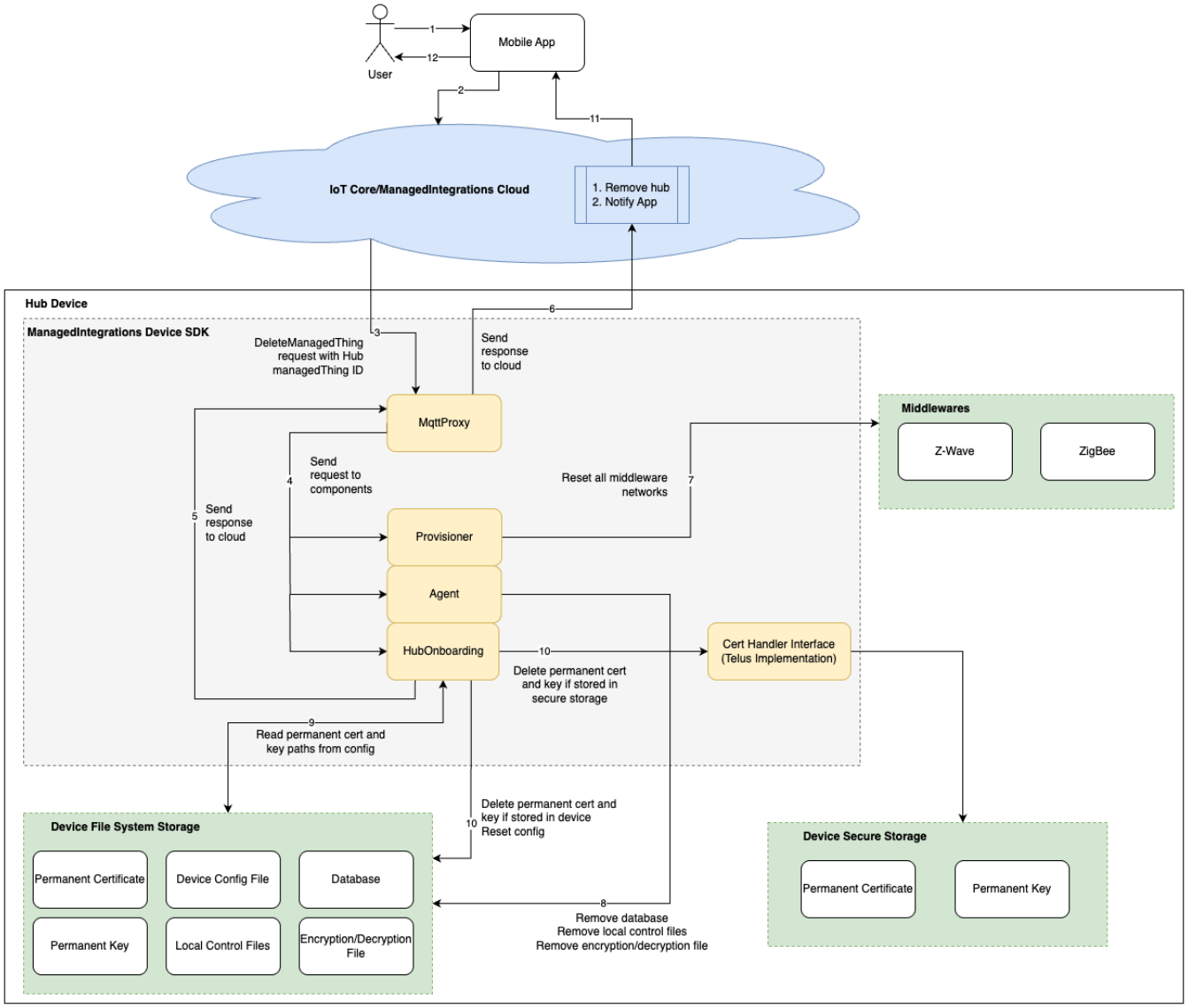
Cloud-side actions:

- Remove all cloud resources associated with the hub
- Complete disconnection from the previous account

Customers typically initiate hub offboarding when:

- Changing the hub's associated account
- Replacing an existing hub with a new device

The process ensures a clean, secure transition between hub configurations, allowing seamless device management and account flexibility.



Prerequisites

- You must have a hub that is onboarded. For instructions, see Hub onboarding setup.
- In the `iotmi_config.json` file located at **/data/aws/iotmi/config/**, verify that `iot_provisioning_state` shows `PROVISIONED`.
- Confirm that the permanent certificates and keys referenced in the `iotmi_config.json` exist in their specified paths.
- Ensure that HubOnboarding, Agent, Provisioner, and MQTT proxy are correctly configured and running.

- Verify that the hub has no child devices. Use the DeleteManagedThing API to remove all child devices before proceeding.

Hub SDK offboard process

Follow these steps to offboard the hub:

Retrieve hub_managed_thing ID

The `iotmi_config.json` file is used to store the managed thing ID for a Managed Integrations hub. This identifier is a critical piece of information that allows the hub to communicate with the AWS IoT Managed Integrations service. The managed thing ID is stored within the **rw** (read-write) section of the JSON file, under the `managed_thing_id` field. This is seen in the following sample configuration:

```
{
      "ro": {
          "iot_provisioning_method": "FLEET_PROVISIONING",
          "iot_claim_cert_path": "PATH",
          "iot_claim_pk_path": "PATH",
          "UPC": "UPC",
          "sh_endpoint_url": "ENDPOINT_URL",
          "SN": "SN",
          "fp_template_name": "TEMPLATENAME"
      },
      "rw": {
          "iot_provisioning_state": "PROVISIONED",
          "client_id": "ID",
          "managed_thing_id": "ID",
          "iot_permanent_cert_path": "CERT_PATH",
          "iot_permanent_pk_path": "KEY",
          "metadata": {
              "last_updated_epoch_time": 1747766125
          }
      }
}
```

Send command to offboard hub

Use your account credentials and run the command with the `managed_thing_id` retrieved in the previous section:

```
aws iot-managed-integrations delete-managed-thing \
    --identifier HUB_MANAGED_THING_ID
```

Verify hub was offboarded

Use your account credentials and run the command with the `managed_thing_id` retrieved in the previous section:

```
aws iot-managed-integrations get-managed-thing \
    --identifier HUB_MANAGED_THING_ID
```

Success and failure scenarios

Success scenario

If the command to offboard the hub was successful, the following sample response is expected:

```
{
      "Message" : "Managed Thing resource not found."
}
```

Additionally, the following sample `iotmi_config.json` would be observed if the hub offboarding command was successful. Verify that the **rw** section contains only `iot_provisioning_state` and optionally **metadata**. The absence of **metadata** is acceptable. `iot_provisioning_state` must be **NOT_PROVISIONED**.

```
{
      "ro": {
          "iot_provisioning_method": "FLEET_PROVISIONING",
          "iot_claim_cert_path": "PATH",
          "iot_claim_pk_path": "PATH",
          "UPC": "1234567890101",
          "sh_endpoint_url": "ENDPOINT_URL",
          "SN": "1234567890101",
          "fp_template_name": "test-template"
      },
      "rw": {
          "iot_provisioning_state": "NOT_PROVISIONED",
          "metadata": {
              "last_updated_epoch_time": 1747766125
          }
```

```
      }
}
```

Failure scenario

If the command to offboard the hub was unsuccessful, the following sample response is expected:

```
{
      "Arn" : "ARN",
      "CreatedAt" : 1.748968266655E9,
      "Id" : "ID",
      "ProvisioningStatus" : "DELETE_IN_PROGRESS",
      "Role" : "CONTROLLER",
      "SerialNumber" : "SERIAL_NO",
      "Tags" : { },
      "UniversalProductCode" : "UPC",
      "UpdatedAt" : 1.748968272107E9
}
```

- If **ProvisioningStatus** is `DELETE_IN_PROGRESS`, follow instructions in Hub recovery.
- If **ProvisioningStatus** is not `DELETE_IN_PROGRESS`, the command to offboard the hub either failed in the Managed Integrations cloud, or was not received by Managed Integrations cloud. Follow instructions in Hub recovery.
- If offboarding was unsuccessful, your `iotmi_config.json` file will look like the sample file below.

```
{
      "ro": {
          "iot_provisioning_method": "FLEET_PROVISIONING",
          "iot_claim_cert_path": "PATH",
          "iot_claim_pk_path": "PATH",
          "UPC": "123456789101",
          "sh_endpoint_url": "ENDPOINT_URL",
          "SN": "123456789101",
          "fp_template_name": "test-template"
      },
      "rw": {
          "iot_provisioning_state": "PROVISIONED",
          "client_id": "ID",
          "managed_thing_id": "ID",
```

```
            "iot_permanent_cert_path": "PATH",
            "iot_permanent_pk_path": "PATH",
            "metadata": {
                "last_updated_epoch_time": 1747766125
            }
        }
}
```

(Optional) After offboarding Hub SDK

> **Important**
>
> The following scenarios list optional actions to take after offboarding Hub SDK failed, or if you want to re-onboard your hub after offboarding.

Re-onboard

If offboarding was successful, onboard your Hub SDK following Step 3: Create a managed thing (fleet provisioning), and the rest of the onboard process.

Hub recovery

Device hub offboarding success and Cloud offboarding fails

If GetManagedThing API call doesn't return `Managed Thing resource not found` message, but the file `iotmi_config.json` is offboarded. See Success scenario for a sample json file.

To recover from this scenario, see Force deletion.

Device hub offboarding fails

This scenario is when the file `iotmi_config.json` is not offboarded correctly. See Failure scenario for a sample json file.

To recover from this scenario, see Force deletion. If `iotmi_config.json` is still not offboarded, the hub has to be factory reset.

Device hub offboarding and Cloud offboarding fails

In this scenario, `iotmi_config.json` is still not offboarded, and hub status is either `ACTIVATED`, or `DISCOVERED`.

To recover from this scenario, see Force deletion. If force deletion fails, or `iotmi_config.json` is still not offboarded, the hub has to be factory reset.

Hub is offline and hub status is DELETE_IN_PROGRESS

In this scenario, the hub is offline and cloud receives an offboarding command.

To recover from this scenario, see Force deletion.

Force deletion

To delete cloud resources without a successful device hub offboarding, follow these steps. This operation may result in inconsistency between the cloud and device states, potentially causing issues with future operations.

Call DeleteManagedThing API with the hub's `managed_thing_id` and the **force** parameter:

```
aws iot-managed-integrations delete-managed-thing \
  --identifier HUB_MANAGED_THING_ID \
  --force
```

Next, call the GetManagedThing API and verify that it returns `Managed Thing resource not found`. This confirms that the cloud resources are deleted.

> **Note**
>
> This approach is not recommended, as it can lead to inconsistencies between the cloud and device states. It's generally better to ensure a successful device hub offboarding before attempting to delete the cloud resources.

Protocol-specific middleware

> **Important**
>
> **The documentation and code provided here describes a reference implementation of the middleware. It is not provided to you as part of the SDK.**

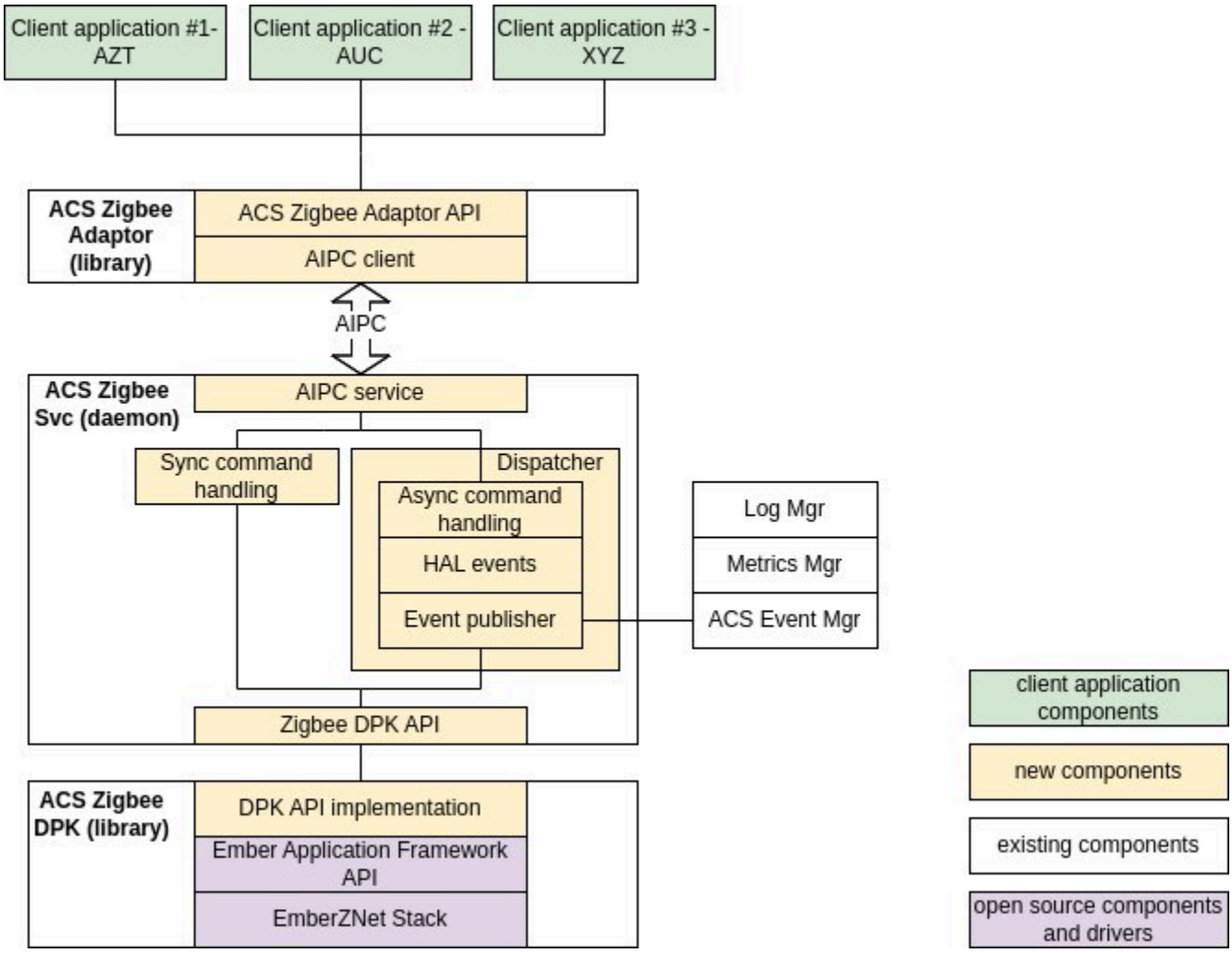
The protocol-specific middleware has a critical role of interacting with the underlying protocol stacks. Both device onboarding and device control components of the managed integrations Hub SDK use it to interact with the end device.

The middleware performs the following functions.

- Abstracts the APIs from the device protocol stacks from different vendors by providing a common set of APIs.
- Provides software execution management such as thread scheduler, event queue management, and data cache.

Middleware architecture

The block diagram below represents the architecture of the Zigbee middleware. The architecture of middlewares of other protocols like Z-Wave is also similar.

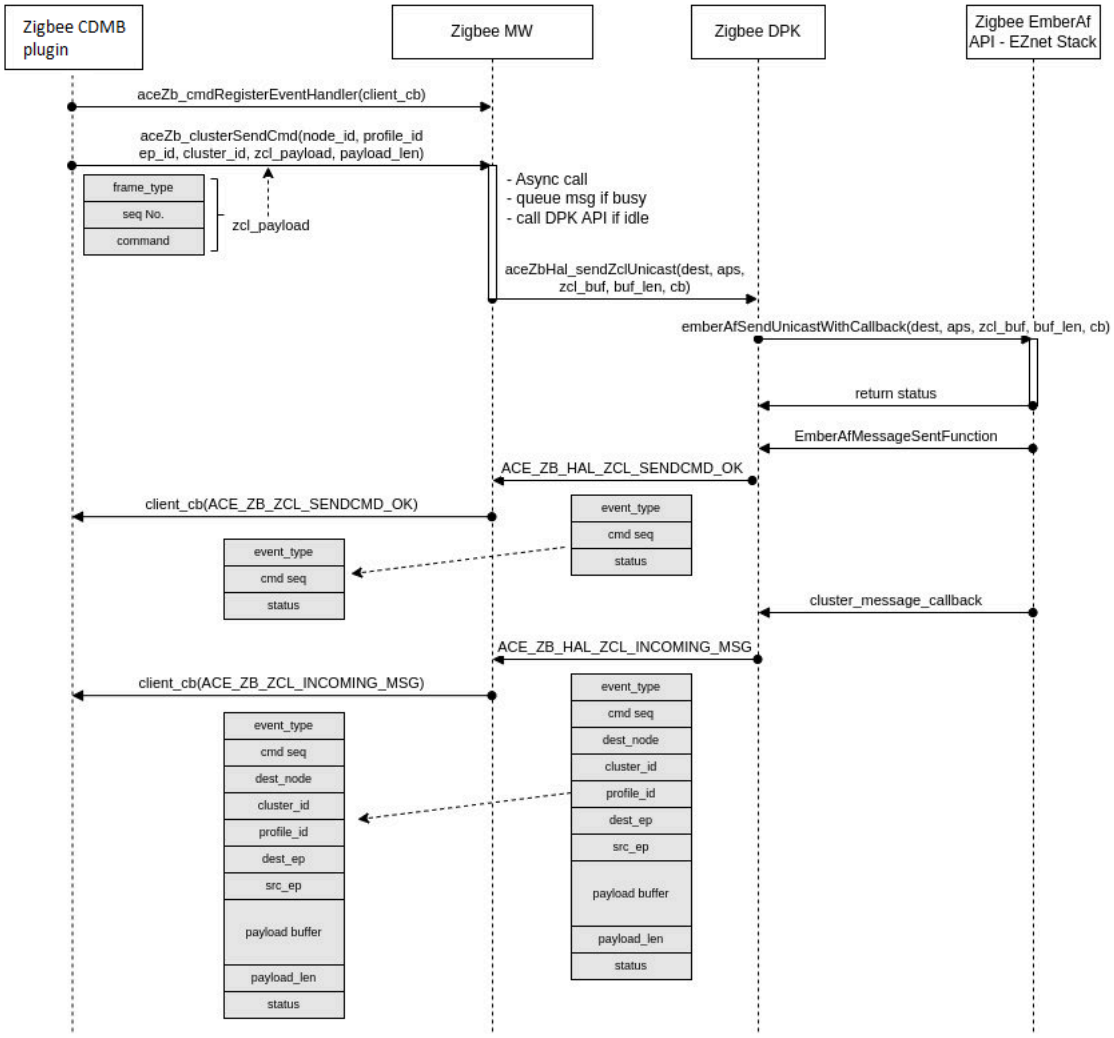


The protocol-specific middleware has three main components.

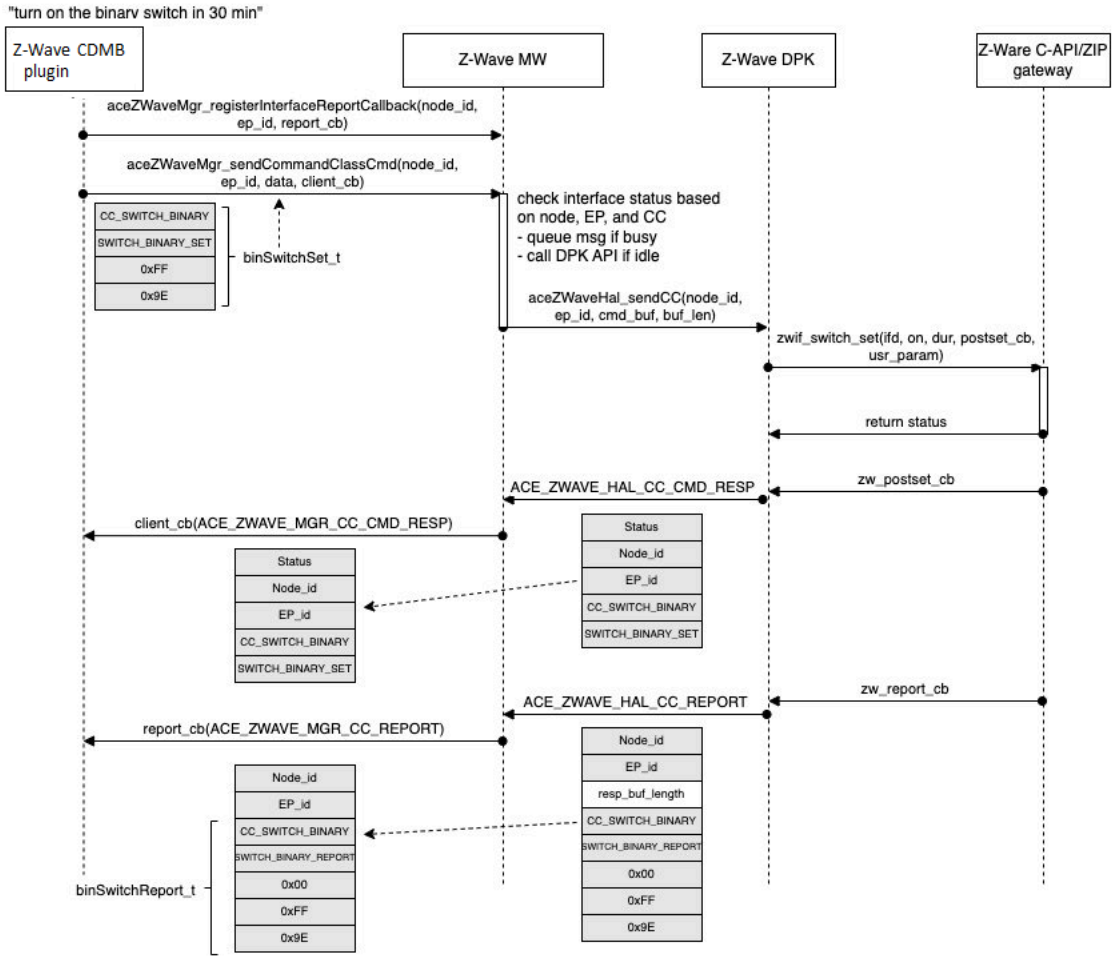
- **ACS Zigbee DPK**: The Zigbee Device Porting Kit (DPK) is used to provide abstraction from the underlying hardware and operating system, thereby enabling portability. Basically this can be considered as the hardware abstraction layer (HAL), which provides a common set APIs to control and communicate with the Zigbee radios from different vendors. The Zigbee middleware contains DPK API implementation for the Silicon Labs Zigbee Application framework.
- **ACS Zigbee Service**: The Zigbee service runs as a dedicated daemon. It includes an API handler serving the API calls from client applications through the IPC channels. AIPC is used as the IPC channel between Zigbee adaptor and Zigbee service. It provides other functionalities like handling both async/sync commands, handling events from the HAL, and using ACS Event Manager for event registering/publishing.
- **ACS Zigbee Adaptor**: The Zigbee adaptor is a library running within the application process (in this case, the application is the CDMB plugin). The Zigbee adaptor provides a set of APIs which are consumed by client applications like CDMB/Provisioner protocol plugins to control and communicate with the end device.

End-to-end middleware command flow example

Here is an example of the command flow through the Zigbee middleware.



Here is an example of the command flow through the Z-Wave middleware.



Protocol-specific middleware code organization

This section contains information about the location of the code for each component inside the `IotManagedIntegrationsDeviceSDK-Middleware` repository. The following is an example of the folder structure in this repository.

```
./IotManagedIntegrationsDeviceSDK-Middleware
|— greengrass
|— example-iot-ace-dpk
|— example-iot-ace-general
|— example-iot-ace-project
|— example-iot-ace-z3-gateway
|— example-iot-ace-zware
|— example-iot-ace-zwave-mw
```

Topics

- Zigbee middleware code organization
- Z-Wave middleware code organization

Zigbee middleware code organization

The following shows the Zigbee reference middleware code organization.

Topics

- ACS Zigbee DPK
- Silicon Labs Zigbee SDK
- ACS Zigbee Service
- ACS Zigbee Adaptor

ACS Zigbee DPK

The code for Zigbee DPK is located inside the directory listed in the example below:

```
./IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-dpk/example/dpk/ace_hal/
|– common
|–   |– fxnDbusClient
|–   |– include
|– kvs
|– log
|– wifi
|–   |– include
|–   |– src
|–   |– wifid
|–       |– fxnWifiClient
|–       |– include
|– zibgee
|–   |– include
|–   |– src
|–   |– zigbeed
|–       |– ember
|–       |– include
|– zwave
|–   |– include
|–   |– src
```

```
|—   |— zwaved
|—       |— fxnZwaveClient
|—       |— include
|—       |— zware
```

Silicon Labs Zigbee SDK

The Silicon Labs SDK is presented inside the `IotManagedIntegrationsDeviceSDK-Middleware/`*`example`*`-iot-ace-z3-gateway` folder. This ACS Zigbee DPK layer is implemented for this Silicon Labs SDK.

```
./IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-zz3-gateway/
|— autogen
|— config
|— gecko_sdk_4.3.2
|—   |— platform
|—   |— protocol
|—   |— util
```

ACS Zigbee Service

The code for the Zigbee Service is located inside the `IotManagedIntegrationsDeviceSDK-Middleware/`*`example`*`-iot-ace-general/middleware/zigbee/` folder. The `src` and `include` subfolders at this location contain all the files related to the ACS Zigbee service.

```
IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-general/middleware/zigbee/
src/
|— zb_alloc.c
|— zb_callbacks.c
|— zb_database.c
|— zb_discovery.c
|— zb_log.c
|— zb_main.c
|— zb_region_info.c
|— zb_server.c
|— zb_svc.c
|— zb_svc_pwr.c
|— zb_timer.c
|— zb_util.c
|— zb_zdo.c
|— zb_zts.c
```

```
IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-general/middleware/zigbee/
include/
|– init.zigbeeservice.rc
|– zb_ace_log_uml.h
|– zb_alloc.h
|– zb_callbacks.h
|– zb_client_aipc.h
|– zb_client_event_handler.h
|– zb_database.h
|– zb_discovery.h
|– zb_log.h
|– zb_region_info.h
|– zb_server.h
|– zb_svc.h
|– zb_svc_pwr.h
|– zb_timer.h
|– zb_util.h
|– zb_zdo.h
|– zb_zts.h
```

ACS Zigbee Adaptor

The code for the ACS Zigbee Adaptor is located inside the `IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-general/middleware/zigbee/api` folder. The `src` and `include` subfolders at this location contain all the files related to the ACS Zigbee Adaptor library.

```
IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-general/middleware/zigbee/
api/src/
|– zb_client_aipc.c
|– zb_client_api.c
|– zb_client_event_handler.c
|– zb_client_zcl.c
IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-general/middleware/zigbee/
api/include/
|– ace
|–   |– zb_adapter.h
|–   |– zb_command.h
|–   |– zb_network.h
|–   |– zb_types.h
|–   |– zb_zcl.h
|–   |– zb_zcl_cmd.h
|–   |– zb_zcl_color_control.h
```

```
|—   |— zb_zcl_hvac.h
|—   |— zb_zcl_id.h
|—   |— zb_zcl_identify.h
|—   |— zb_zcl_level.h
|—   |— zb_zcl_measure_and_sensing.h
|—   |— zb_zcl_onoff.h
|—   |— zb_zcl_power.h
```

Z-Wave middleware code organization

The following shows the Z-wave reference middleware code organization.

Topics

- ACS Z-Wave DPK
- Silicon Labs ZWare and Zip Gateway
- ACS Z-Wave Service
- ACS Z-Wave Adaptor

ACS Z-Wave DPK

The code for Z-Wave DPK is located inside the `IotManagedIntegrationsDeviceSDK-Middleware/`*`example`*`-iot-ace-dpk/`*`example`*`/dpk/ace_hal/zwave` folder.

```
./IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-dpk/example/dpk/ace_hal/
|— common
|—   |— fxnDbusClient
|—   |— include
|— kvs
|— log
|— wifi
|—   |— include
|—   |— src
|—   |— wifid
|—       |— fxnWifiClient
|—       |— include
|— zibgee
|—   |— include
|—   |— src
|—   |— zigbeed
|—       |— ember
```

```
|–       |– include
|– zwave
|–   |– include
|–   |– src
|–   |– zwaved
|–       |– fxnZwaveClient
|–       |– include
|–       |– zware
```

Silicon Labs ZWare and Zip Gateway

The code for the Silicon labs ZWare and Zip Gateway is located inside the `IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-z3-gateway` folder. This ACS Z-Wave DPK layer is implemented for Z-Wave C-APIs and Zip gateway.

```
./IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-z3-gateway/
|– autogen
|– config
|– gecko_sdk_4.3.2
|–   |– platform
|–   |– protocol
|–   |– util
```

ACS Z-Wave Service

The code for the Z-Wave Service is located inside the folder listed in the `IoTmanagedintegrationsMiddlewares/exampleiot-ace-zwave-mw/` folder. The `src` and `include` folders at this location contain all the files related to the ACS Z-Wave service.

```
IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-zwave-mw/src/
|– zwave_mgr.c
|– zwave_mgr_cc.c
|– zwave_mgr_ipc_aipc.c
|– zwave_svc.c
|– zwave_svc_dispatcher.c
|– zwave_svc_hsm.c
|– zwave_svc_ipc_aipc.c
|– zwave_svc_main.c
|– zwave_svc_publish.c
IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-zwave-mw/include/
|– ace
|–   |– zwave_common_cc.h
```

```
|—   |— zwave_common_cc_battery.h
|—   |— zwave_common_cc_doorlock.h
|—   |— zwave_common_cc_firmware.h
|—   |— zwave_common_cc_meter.h
|—   |— zwave_common_cc_notification.h
|—   |— zwave_common_cc_sensor.h
|—   |— zwave_common_cc_switch.h
|—   |— zwave_common_cc_thermostat.h
|—   |— zwave_common_cc_version.h
|—   |— zwave_common_types.h
|—   |— zwave_mgr.h
|—   |— zwave_mgr_cc.h
|— zwave_log.h
|— zwave_mgr_internal.h
|— zwave_mgr_ipc.h
|— zwave_svc_hsm.h
|— zwave_svc_internal.h
|— zwave_utils.h
```

ACS Z-Wave Adaptor

The code for the ACS Zigbee Adaptor is located inside the `IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-zwave-mw/cli/` folder. The `src` and `include` folder at this location contain all the files related to the ACS Z-Wave Adaptor library.

```
IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-zwave-mw/cli/
|— include
|—   |— zwave_cli.h
|— src
|—   |— zwave_cli.yaml
|—   |— zwave_cli_cc.c
|—   |— zwave_cli_event_monitor.c
|—   |— zwave_cli_main.c
|—   |— zwave_cli_net.c
```

Integrate middleware with SDK

The middleware integration on the new hub is discussed in the following sections.

Topics

- Device porting kit (DPK) API integration

- Reference implementation and code organization

Device porting kit (DPK) API integration

To integrate any chipset vendor SDK with the middleware, a standard API interface is provided by the DPK (Device porting kit) layer of the middle. The managed integrations service providers or ODMs need to implement these APIs based on the vendor SDK supported by the Zigbee/Z-wave/ Wi-Fi chipsets used on their IoT Hubs.

Reference implementation and code organization

Except the middleware, all other Device SDK components, such as the managed integrations Device Agent and Common Data Model Bridge (CDMB) can be used without any modifications and only need to be cross compiled.

The implementation of the middleware is based on the Silicon Labs SDK for Zigbee and Z-Wave. If the Z-Wave and Zigbee chipsets used in the new hub are supported by the Silicon Labs SDK present in the middleware, then the reference middleware can be used without any modifications. You only need to cross-compile the middleware and it can then be run on the new hub.

DPK (Device porting kit) APIs for Zigbee can be found in `acehal_zigbee.c`, and the reference implementation of the DPK APIs is present inside the `zigbee` folder.

```
IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-dpk/example/dpk/ace_hal/
zigbee/
|— CMakeLists.txt
|— include
|—   |— zigbee_log.h
|— src
|—   |— acehal_zigbee.c
|— zigbeed
|—   |— CMakeLists.txt
|—   |— ember
|—   |—   |— ace_ember_common.c
|—   |—   |— ace_ember_ctrl.c
|—   |—   |— ace_ember_hal_callbacks.c
|—   |—   |— ace_ember_network_creator.c
|—   |—   |— ace_ember_power_settings.c
|—   |—   |— ace_ember_zts.c
|—   |— include
|—   |—   |— zbd_api.h
```

```
|—   |—   |— zbd_callbacks.h
|—   |—   |— zbd_common.h
|—   |—   |— zbd_network_creator.h
|—   |—   |— zbd_power_settings.h
|—   |—   |— zbd_zts.h
```

DPK APIs for Z-Wave can be found in the `acehal_zwave.c` and reference implementation of the DPK APIs is present inside the `zwaved` folder.

```
IotManagedIntegrationsDeviceSDK-Middleware/example-iot-ace-dpk/example/dpk/ace_hal/
zwave/
|— CMakeLists.txt
|— include
|—   |— zwave_log.h
|— src
|—   |— acehal_zwave.c
|— zwaved
|—   |— CMakeLists.txt
|—   |— fxnZwaveClient
|—   |—   |— zwave_client.c
|—   |—   |— zwave_client.h
|—   |— include
|—   |—   |— zwaved_cc_intf_api.h
|—   |—   |— zwaved_common_utils.h
|—   |—   |— zwaved_ctrl_api.h
|—   |— zware
|—   |—   |— ace_zware_cc_intf.c
|—   |—   |— ace_zware_common_utils.c
|—   |—   |— ace_zware_ctrl.c
|—   |—   |— ace_zware_debug.c
|—   |—   |— ace_zware_debug.h
|—   |—   |— ace_zware_internal.h
```

As a starting point to implement the DPK layer for a different vendor SDK, the reference implementation can be used and modified. Following two modifications will be needed to support a different vendor SDK:

1. Replace the current vendor SDK with the new vendor SDK in the repository.
2. Implement the middleware DPK (Device porting kit) APIs according to the new vendor SDK.

Managed Integrations End device SDK

Build an IoT platform that connects smart devices to Managed Integrations and processes commands through a unified control interface. The End device SDK integrates with your device firmware and provides simplified setup with the SDK edge components, and secure connectivity to AWS IoT Core and AWS IoT Device Management. Download the End device SDK version 1.2.0.

This guide describes how to implement the End device SDK in your firmware. Review the architecture, components, and integration steps to start building your implementation.

Topics

- What is the End device SDK?
- End device SDK architecture and components
- Provisionee
- Over-the-Air updates
- Data model code generator
- Low level C-Function APIs
- Feature and device interactions in Managed Integrations
- Get started with End device SDK

What is the End device SDK?

What is the End device SDK?

The End device SDK is a collection of source code, libraries, and tools provided by AWS IoT. Built for resource-constrained environments, the SDK supports devices with as little as 512 KB RAM and 4 MB flash memory, such as cameras and air purifiers running on embedded Linux and real-time operating systems (RTOS). Download the End device SDK version 1.2.0.

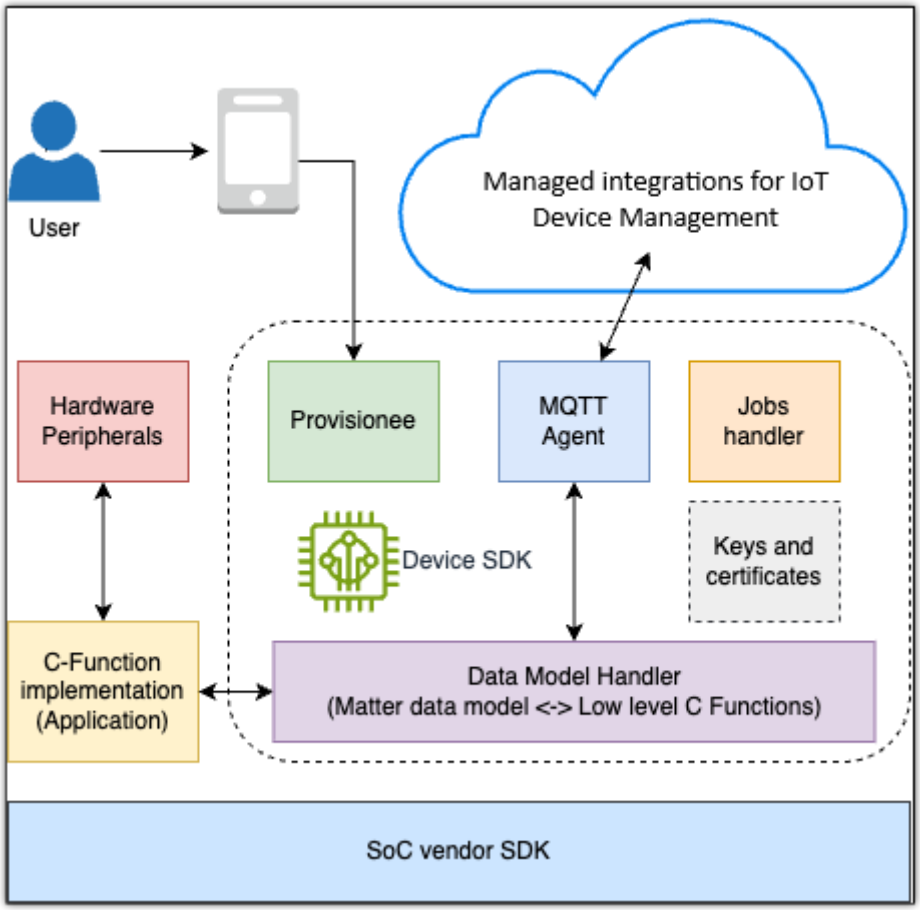
Core components

The SDK combines an MQTT agent for cloud communication, a jobs handler for task management, and a Managed Integrations, Data Model Handler. These components work together to provide secure connectivity and automated data translation between your devices and Managed Integrations.

For detailed technical requirements, see the Technical reference.

End device SDK architecture and components

This section describes the End device SDK architecture and how its components interact with your low level C-Functions. The following diagram illustrates the core components and their relationships in the SDK framework.



End device SDK components

The End device SDK architecture contains these components for Managed Integrations feature integration:

Provisionee

Creates device resources in the Managed Integrations cloud, including device certificates and private keys for secure MQTT communication. These credentials establish trusted connections between your device and Managed Integrations.

MQTT Agent

Manages MQTT connections through a thread-safe C client library. This background process handles command queues in multi-threaded environments, with configurable queue sizes for memory-constrained devices. Messages route through Managed Integrations for processing.

Jobs handler

Processes over-the-air (OTA) updates for device firmware, security patches, and file delivery. This built-in service manages software updates for all registered devices.

Data Model Handler

Translates operations between Managed Integrations and your Low Level C-Functions using AWS' implementation of the Matter Data Model. For more information, see the Matter documentation on *GitHub*.

Keys and certificates

Manages cryptographic operations through the PKCS #11 API, supporting both hardware security modules and software implementations like corePKCS11. This API handles certificate operations for components such as the Provisionee and MQTT Agent during TLS connections.

Provisionee

The provisionee is a component of Managed Integrations that enables fleet provisioning by claim. With the provisionee, you securely provision your devices. The SDK creates the necessary resources for device provisioning, which includes the device certificate and private keys that are obtained from the Managed Integrations cloud. When you want to provision your devices, or if there are any changes that can require you to re-provision your devices, you can use the provisionee.

Topics

- Provisionee workflow
- Set environment variables
- Register a custom endpoint
- Create a provisioning profile
- Create a managed thing
- SDK user Wi-Fi provisioning

- Fleet provisioning by claim
- Managed thing capabilities

Provisionee workflow

The process requires setup on both cloud and device sides. Customers configure cloud requirements like custom endpoints, provisioning profiles, and managed things. At first device power-on, the provisionee:

1. Connects to the Managed Integrations endpoint using a claim certificate
2. Validates device parameters through fleet provisioning hooks
3. Obtains and stores a permanent certificate and private key on the device
4. The device uses the permanent certificate to reconnect
5. Discovers and uploads device capabilities to Managed Integrations

After successful provisioning, the device communicates directly with Managed Integrations. The provisionee activates only for re-provisioning tasks.

Set environment variables

Set the following AWS credentials in your cloud environment:

```
$ export AWS_ACCESS_KEY_ID=YOUR-ACCOUNT-ACCESS-KEY-ID
$ export AWS_SECRET_ACCESS_KEY=YOUR-ACCOUNT-SECRET-ACCESS-KEY
$ export AWS_DEFAULT_REGION=YOUR-DEFAULT-REGION
```

Register a custom endpoint

Use the RegisterCustomEndpoint API command in your cloud environment to create a custom endpoint for device-to-cloud communication.

```
aws iot-managed-integrations register-custom-endpoint
```

Example response

```
{ "EndpointAddress":"[ACCOUNT-PREFIX]-ats.iot.AWS-REGION.amazonaws.com" }
```

> **Note**
>
> Store the endpoint address for configuring provisioning parameters. use the `GetCustomEndpoint` API, to return endpoint information. For more information, see GetCustomEndpoint API, and RegisterCustomEndpoint API in the *Managed Integrations API Reference Guide*.

Create a provisioning profile

Create a provisioning profile that defines your fleet provisioning method. Run the CreateProvisioningProfile API in your cloud environment to return a claim certificate and private key for device authentication:

```
aws iot-managed-integrations create-provisioning-profile \
--provisioning-type "FLEET_PROVISIONING" \
--name "PROVISIONING-PROFILE-NAME"
```

Example response

```
{ "Arn":"arn:aws:iot-managed-integrations:AWS-REGION:YOUR-ACCOUNT-ID:provisioning-
profile/PROFILE_NAME",
    "ClaimCertificate":"string",
    "ClaimCertificatePrivateKey":"string",
    "Name":"ProfileName",
    "ProvisioningType":"FLEET_PROVISIONING"}
```

You can implement the corePKCS11 platform abstraction library (PAL) to make the corePKCS11 library work with your device. The corePKCS11 PAL ports must provide a location to store the claim certificate and private key. Using this feature, you can securely store the device's private key and certificate. You can store the private key and certificate on a hardware security module (HSM) or a trusted platform module (TPM).

Create a managed thing

Register your device with Managed Integrations cloud by using the CreateManagedThing API. Include the serial number (SN) and universal product code (UPC) of your device:

```
aws iot-managed-integrations create-managed-thing –role DEVICE \
  --authentication-material-type WIFI_SETUP_QR_BAR_CODE \
  --authentication-material "SN:DEVICE-SN;UPC:DEVICE-UPC;"
```

The following displays a sample API response.

```
{
   "Arn":"arn:aws:iot-managed-integrations:AWS-REGION:ACCOUNT-ID:managed-
thing/59d3c90c55c4491192d841879192d33f",
   "CreatedAt":1.730960226491E9,
   "Id":"59d3c90c55c4491192d841879192d33f"
}
```

The API returns the **Managed thing ID** that can be used for provisioning validation. You will need to provide the device serial number (SN) and universal product code (UPC), which are matched with the approved managed thing during the provisioning transaction. The transaction returns a result similar to the following:

```
/**
 * @brief Device info structure.
 */
typedef struct iotmiDev_DeviceInfo
{
    char serialNumber[ IOTMI_DEVICE_MAX_SERIAL_NUMBER_LENGTH + 1U ];
    char universalProductCode[ IOTMI_DEVICE_MAX_UPC_LENGTH + 1U ];
    char internationalArticleNumber[ IOTMI_DEVICE_MAX_EAN_LENGTH + 1U ];
} iotmiDev_DeviceInfo_t;
```

SDK user Wi-Fi provisioning

Device manufacturers and solution providers have their own proprietary Wi-Fi provisioning service for receiving and configuring Wi-Fi credentials. The Wi-Fi provisioning service involves using dedicated mobile apps, Bluetooth Low Energy (BLE) connections, and other proprietary protocols to securely transfer Wi-Fi credentials for the initial setup process.

The consumer of the End device SDK must implement the Wi-Fi provisioning service and the device can connect to a Wi-Fi network.

Alternatively, the End device SDK can be built using the `IOTMI_USE_WSS_PROVISIONEE` build flag to enable WiFi Simple Setup (WSS) provisioning. WSS uses the Hub SDK's provisioner to

automatically provision WiFi credentials, providing an alternative to proprietary WiFi provisioning methods. For more information, see the section called “WiFi Simple Setup to onboard and operate devices”.

Fleet provisioning by claim

Using the provisionee, the end user can provision a unique certificate and register it with Managed Integrations using provisioning by claim.

The client ID can be acquired either from the provisioning template response or the device certificate `<common name>“_”<serial number>`

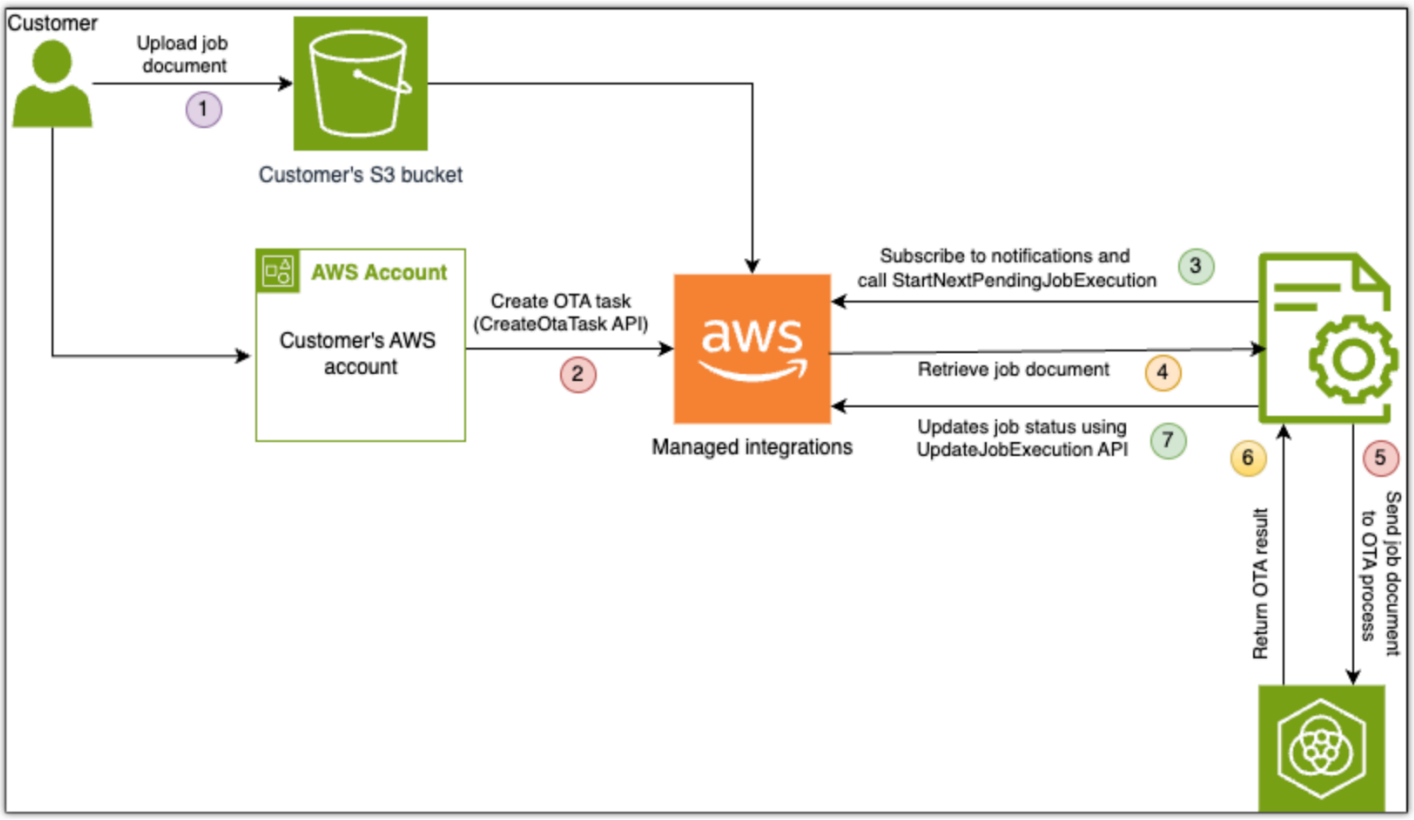
Managed thing capabilities

The provisionee discovers the managed thing capabilities, then uploads the capabilities to Managed Integrations. It makes the capabilities available to apps and other services to access. Devices, other web clients, and services can update the capabilities by using MQTT and the reserved MQTT topic, or HTTP using the REST API.

Over-the-Air updates

OTA architecture overview

The Over-the-Air (OTA) update process involves several components working together to deliver firmware updates to your devices. The following diagram illustrates how an OTA update request is handled through the interaction between End device SDK, Hub SDK and the feature.



The OTA update architecture consists of the following components:

- **Customer**: Uploads job documents to a S3 bucket and initiates updates via API
- **OTA Service**: Handles job creation, validation, and management
- **AWS IoT Jobs**: Manages job execution and delivery to devices
- **Devices**: Receive and apply updates using Harmony SDK

Prerequisites for OTA updates

Before creating Over-the-Air (OTA) tasks, you need to complete the following configuration tasks in your environment.

Configure Amazon S3 access

To enable OTA updates, you must upload your job documents to an Amazon Simple Storage Service (Amazon S3) bucket and configure the appropriate access permissions for the bucket.

To configure Amazon S3 access

1. Determine which S3 bucket you want to use for OTA updates.
2. Upload your OTA job document to the bucket.
3. Configure the bucket policy to allow AWS IoT Managed Integrations to access your job documents in the bucket:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "PolicyForS3JobDocument",
      "Effect": "Allow",
      "Principal": {
        "Service": "iotmanagedintegrations.amazonaws.com"
      },
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::amzn-s3-demo-bucket/*",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "111122223333"
        }
      }
    }
  ]
}
```

Update the preceding policy with the correct values for your environment. In the `Resource` element, replace *amzn-s3-demo-bucket* with the name of your bucket. To restrict access to a specific location in the bucket, also add the prefix for the path to the location—for example, `arn:aws:s3:::amzn-s3-demo-bucket/my-OTA-job-documents/*`. In the `Condition` block, replace *111122223333* with the account ID for your AWS account.

Implement Over-the-Air (OTA) tasks

You can create OTA tasks in two ways, depending on your update requirements and device targeting strategy:

One-Time OTA task updates

A one-time OTA task contains a static list of targets (`ManagedThings`) to perform OTA updates. You can add up to 100 targets at a time. The workflow uses AWS IoT Jobs with Fleet Indexing while maintaining the managed integrations abstraction layer.

Use the following example to create a one-time OTA task:

```
aws iotmanagedintegrations create-ota-task \
  --description "One-time OTA update" \
  --s3-url "s3://test-job-document-bucket/ota-job-document.json" \
  --protocol HTTP \
  --target ["arn:aws:iotmanagedintegrations:region:account id:managed-thing/managed
 thing id"] \
  --ota-mechanism PUSH \
  --ota-type ONE_TIME \
  --client-token "foo" \
  --tags '{"key1":"foo","key2":"foo"}'
```

Continuous OTA task updates

The OTA (Over-the-Air) grouping workflow enables you to deploy firmware updates to groups of devices based on specific attributes, using AWS IoT Jobs with Fleet Indexing while maintaining the managed integrations abstraction layer. Continuous OTA tasks use a query string instead of specific targets. All devices that match the query criteria undergo OTA updates, and the query criteria is continually re-evaluated. The matching targets will have job deployments.

Configure prerequisites

Before creating continuous OTA tasks, complete these prerequisites:

1. Create a managed thing by calling the CreateManagedThing API and perform fleet provisioning.
2. Add metadata attributes to your managed things for query targeting.

Add attributes and metadata to `ManagedThing` using the UpdateManagedThing API:

```
aws iotmanagedintegrations update-managed-thing \
  --managed-thing-id "YOUR_MANAGED_THING_ID" \
  --meta-data '{"owner":"managedintegrations","version":"1.0"}'
```

Use the following example to create a continuous OTA task:

```
aws iotmanagedintegrations create-ota-task \
  --description "Continuous OTA update" \
  --s3-url "s3://test-job-document-bucket/ota-job-document.json" \
  --protocol HTTP \
  --ota-mechanism PUSH \
  --ota-type CONTINUOUS \
  --client-token "foo" \
  --ota-target-query-string "attributes.owner=managedintegrations" \
  --tags '{"key1":"foo","key2":"foo"}'
```

Understand continuous OTA workflow

The continuous OTA update workflow follows these steps:

1. You update managed things with attributes using the UpdateManagedThing API.
2. Create an OTA job with a query string targeting specific device attributes.
3. The OTA service creates a dynamic Thing Group in AWS IoT Core based on query attributes
4. IoT Jobs executes updates on matching devices
5. You monitor progress via the ListOtaTaskExecutions API or OTA notifications through Kinesis stream (if enabled).

Differences between Managed Integrations OTA and IoT Jobs

The fundamental distinction between Managed Integrations OTA and IoT Jobs lies in **service orchestration and automation**. Managed Integrations OTA provides a **single-service solution** that abstracts away the complexity of multi-service coordination.

What Managed Integrations OTA does automatically:

- **Dynamic Thing Group creation:** Automatically generates AWS IoT Core thing groups based on your query criteria.
- **Target resolution:** Translates query strings (Example: `attributes.owner=managedintegrations`) into actual device targets.
- **Service integration:** Seamlessly coordinates between AWS IoT Core, IoT Jobs, and Fleet Indexing services.

- **Lifecycle management:** Handles the entire OTA workflow from creation to execution monitoring.

What MI OTA eliminates:

- Creating thing groups in AWS IoT Core.
- Adding things to groups.
- Creating IoT Jobs.

Managed Integrations OTA handles all three operations internally based on your query string, automatically discovering devices that match your criteria, creating IoT Jobs under the hood, and orchestrating the complete OTA workflow without requiring you to interact with multiple AWS services directly.

OTA task configurations setup

You can create configurations for OTA updates to control how updates are rolled out to devices, set abort conditions, and configure timeouts.

Example: CreateOtaTaskConfiguration

Use the following example to create an OTA task configuration:

```
aws iotmanagedintegrations create-ota-task-configuration \
  --description "OTA configuration" \
  --name "MyOtaConfig" \
  --push-config '{
    "AbortConfig": {
      "AbortConfigCriteriaList": [
        {
          "Action": "CANCEL",
          "FailureType": "FAILED",
          "MinNumberOfExecutedThings": 1,
          "ThresholdPercentage": 90.0
        }
      ]
    },
    "RolloutConfig": {
      "ExponentialRolloutRate": {
        "BaseRatePerMinute": 1,
```

```
        "IncrementFactor": 3.0,
        "RateIncreaseCriteria": {
          "numberOfNotifiedThings": 1
        }
      },
      "MaximumPerMinute": 1
    },
    "TimeoutConfig": {
      "InProgressTimeoutInMinutes": 100
    }
  }' \
  --client-token "foo"
```

Apply configuration settings to OTA tasks

Once the configuration is created, you'll receive a `taskConfigurationId` that is added to your `CreateOtaTask` request along with additional configurations:

```
aws iotmanagedintegrations create-ota-task \
  --description "OTA with configuration" \
  --s3-url "s3://test-job-document-bucket/ota-job-document.json" \
  --protocol HTTP \
  --target ["arn:aws:iotmanagedintegrations:region:account id:managed-thing/managed
 thing id"] \
  --ota-mechanism PUSH \
  --ota-type ONE_TIME \
  --client-token "foo" \
  --task-configuration-id "ae4f49352c5443369f43ad6c3a7f1580" \
  --ota-scheduling-config '{
    "EndBehavior": "STOP_ROLLOUT",
    "EndTime": "2024-10-23T17:00",
    "StartTime": "2024-10-20T17:00"
  }' \
  --ota-task-execution-retry-config '{
    "RetryConfigCriteria": [
      {
        "FailureType": "FAILED",
        "MinNumberOfRetries": 1
      }
    ]
  }' \
  --tags '{"key1":"foo","key2":"foo"}'
```

Monitor OTA notifications

You can monitor OTA updates using two different methods:

Push notifications through Kinesis Data Streams

When OTA notifications are enabled, update status events are automatically pushed to your Kinesis stream. This provides real-time visibility into the progress of firmware updates across devices.

Monitor with ListOtaTaskExecutions API

You can use the ListOtaTaskExecutions API to manually check the status of OTA updates for your managed things:

```
aws iotmanagedintegrations list-ota-task-executions \
  --task-id "task-123456789" \
  --max-results 25
```

The response provides detailed execution status for each managed thing:

```
{
  "taskExecutionSummaries": [
    {
      "taskExecutionSummary": {
        "executionNumber": 1,
        "lastUpdatedAt": 1634567890,
        "queuedAt": 1634567800,
        "startedAt": 1634567830,
        "status": "SUCCEEDED",
        "retryAttempt": 0
      },
      "managedThingId": "device-001"
    },
    {
      "taskExecutionSummary": {
        "executionNumber": 1,
        "lastUpdatedAt": 1634567920,
        "queuedAt": 1634567800,
        "startedAt": 1634567840,
        "status": "IN_PROGRESS",
        "retryAttempt": 0
      },
```

```
      "managedThingId": "device-002"
    }
  ],
  "nextToken": "NEXT_TOKEN"
}
```

This API allows you to retrieve detailed execution status for each managed thing targeted by a specific OTA task, including timestamps and current status.

Process job documents

When you create an OTA task, the jobs handler runs the following steps on your device. When an update is available, it requests the job document over MQTT.

1. Subscribes to the MQTT notification topics.
2. Calls the StartNextPendingJobExecution API for pending jobs.
3. Receives available job documents.
4. Processes updates based on your specified timeouts.

Using the jobs handler, the application can determine whether to take action immediately or wait until a specified timeout period.

Implement OTA agent

When you receive the job document from managed integrations, you must have an implementation of your own OTA agent that processes the job document, downloads updates, and performs any installation operations. The OTA Agent needs to perform the following steps:

1. Parse job documents for firmware Amazon S3 URLs.
2. Download firmware updates through HTTP.
3. Verify digital signatures.
4. Install validated updates.
5. Call `iotmi\_JobsHandler\_updateJobStatus` with SUCCESS or `FAILED` status.

When your device successfully completes the OTA operation, it must call the `iotmi\_JobsHandler\_updateJobStatus` API with a status of `JobSucceeded` to report a successful job.

```
/**
 * @brief Enumeration of possible job statuses.
 */
typedef enum{
    JobQueued,     /** The job is in the queue, waiting to be processed. */
    JobInProgress, /** The job is currently being processed. */
    JobFailed,     /** The job processing failed. */
    JobSucceeded,  /** The job processing succeeded. */
    JobRejected    /** The job was rejected, possibly due to an error or invalid
 request. */
} iotmi_JobCurrentStatus_t;

/**
 * @brief Update the status of a job with optional status details.
 *
 * @param[in] pJobId Pointer to the job ID string.
 * @param[in] jobIdLength Length of the job ID string.
 * @param[in] status The new status of the job.
 * @param[in] statusDetails Pointer to a string containing additional details about the
 job status.
 *                          This can be a JSON-formatted string or NULL if no details
 are needed.
 * @param[in] statusDetailsLength Length of the status details string. Set to 0 if
 `statusDetails` is NULL.
 *
 * @return 0 on success, non-zero on failure.
 */
int iotmi_JobsHandler_updateJobStatus( const char * pJobId,
                                       size_t jobIdLength,
                                       iotmi_JobCurrentStatus_t status,
                                       const char * statusDetails,
                                       size_t statusDetailsLength );
```

Data model code generator

Learn how to use the code generator for the data model. The generated code can be used to serialize and deserialize the data models that are exchanged between the cloud and the device.

The project repository contains a code generation tool for creating C code data model handlers. The following topics describe the code generator and the workflow.

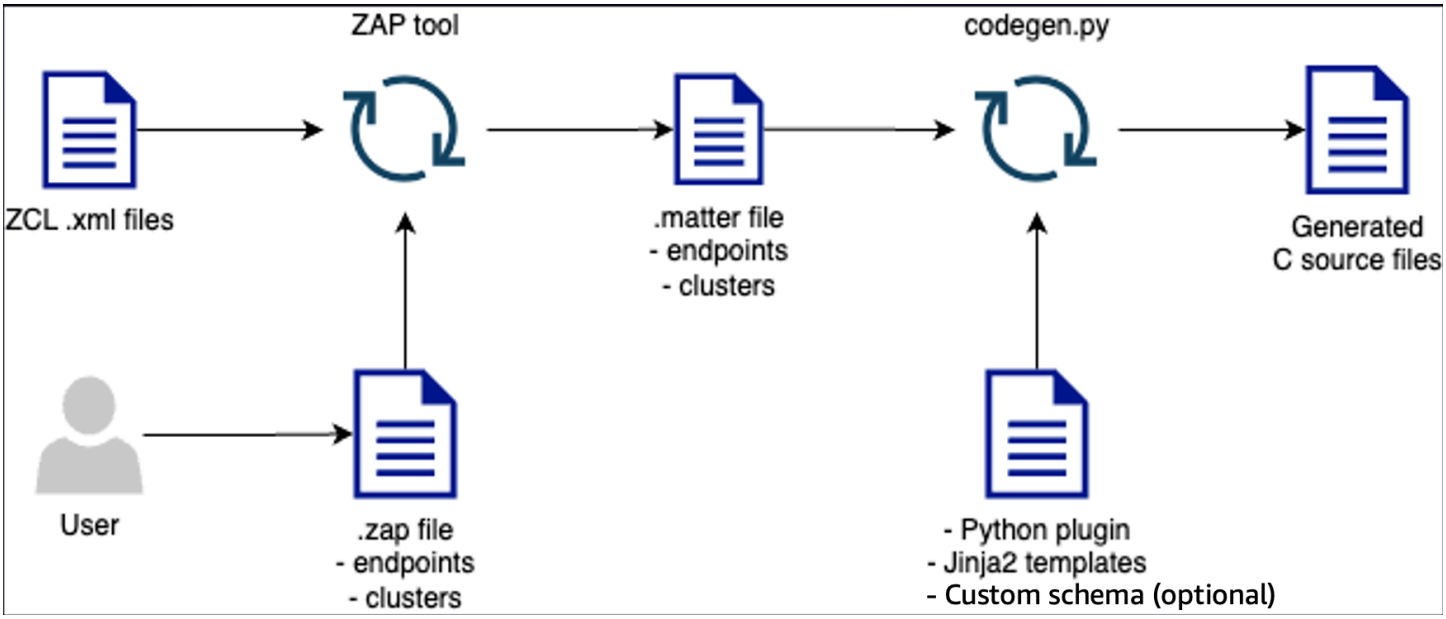
Topics

- Code generation process
- Environment setup
- Generate code for devices

Code generation process

The code generator creates C source files from three primary inputs: AWS' implementation of the Matter Data Model (.matter file) from the Zigbee Cluster Library (ZCL) Advanced Platform, a Python plugin that handles preprocessing, and Jinja2 templates that define code structure. During generation, the Python plugin processes your .matter files by adding global type definitions, organizing data types based on their dependencies, and formatting the information for template rendering.

The following image describes the code generator creating the C source files.



The End device SDK includes Python plugins and Jinja2 templates that work with codegen.py in the connectedhomeip project. This combination generates multiple C files for each cluster based on your .matter file input.

The following subtopics describe these files.

- Python plugin
- Jinja2 templates
- (Optional) Custom schema

Python plugin

The code generator, `codegen.py`, parses the .matter files, and sends the information as Python objects to the plugin. The plugin file `iotmi_data_model.py` preprocesses this data and renders sources with provided templates. Preprocessing includes:

1. Adding information not available from `codegen.py`, such as global types
2. Performing topological sort on data types to establish correct definition order

> **Note**
>
> The topological sort ensures dependent types are defined after their dependencies, regardless of their original order.

Jinja2 templates

The End device SDK provides Jinja2 templates tailored for data model handlers and low level C-Functions.

Jinja2 templates

Template	Generated source	Remarks
`cluster.h.jinja`	`iotmi_device_<cluster>.h`	Creates low level C function header files.
`cluster.c.jinja`	`iotmi_device_<cluster>.c`	Implement and register callback function pointers with the Data Model Handler.
`cluster_type_helpers.h.jinja`	`iotmi_device_type_helpers_<cluster>.h`	Defines function prototypes for data types.
`cluster_type_helpers.c.jinja`	`iotmi_device_type_helpers_<cluster>.c`	Generates data type function prototypes for cluster-specific enumerations, bitmaps, lists, and structures.

Template	Generated source	Remarks
`iot_device_dm_types.h.jinja`	`iotmi_device_dm_types.h`	Defines C data types for global data types.
`iot_device_type_helpers_global.h.jinja`	`iotmi_device_type_helpers_global.h`	Defines C data types for global operations.
`iot_device_type_helpers_global.c.jinja`	`iotmi_device_type_helpers_global.c`	Declares standard data types including boolean, integers, floating point, strings, bitmaps, lists, and structures.

(Optional) Custom schema

End device SDK combines the standardized code generation process with custom schema. This enables extension of Matter Data Model for your devices and device software. Custom schemas can help describe device capabilities for device-to-cloud communication.

For detailed information about managed integrations data models, including format, structure, and requirements, see Managed Integrations data model.

Use `codegen.py` tool to generate C source files for custom schema, as follows:

> **Note**
>
> Each custom cluster requires the same **cluster ID** for the following three files.

- Create custom schema in `JSON` format that provides a representation of clusters for capability reporting to create new custom clusters in the cloud. A sample file is located at `codegen/custom_schemas/custom.SimpleLighting@1.0`.
- Create ZCL (Zigbee Cluster Library) definition file in `XML` format that contains the same information as the custom schema. Use the ZAP tool to generate your Matter IDL files from ZCL XML. A sample file is located at `codegen/zcl/custom.SimpleLighting.xml`.
- The output from ZAP tool is `Matter IDL File (.matter)` and it defines the Matter clusters corresponding to your custom schema. This is the input for `codegen.py` tool to generate C

source files for End device SDK. A sample file is located at `codegen/matter_files/custom-light.matter`.

For detailed instructions on how to integrate custom managed integrations data models into your code generation workflow, see Generate code for devices.

Environment setup

Learn how to configure your environment to use the `codegen.py` code generator.

Topics

- Prerequisites
- Configure your environment

Prerequisites

Install the following items before you configure your environment:

- Git
- Python 3.10 or higher
- Poetry 1.2.0 or higher

Configure your environment

Use the following procedure to configure your environment to use the codegen.py code generator.

1. Download the latest version of the End device SDK from the AWS Management Console.
2. Set up the Python environment. The **codegen** project is python-based and uses Poetry for dependency management.
    - Install project dependencies using poetry in the `codegen` directory:

```
poetry run poetry install --no-root
```

3. Set up your repository.

a. Clone the **connectedhomeip** repository. It uses the `codegen.py` script located in the `connectedhomeip/scripts/` folder for code generation. For more information, see connectedhomeip on *GitHub*.

```
git clone -b v1.4.0.0 https://github.com/project-chip/connectedhomeip.git
```

b. Clone it at the same level as your `IoT-managed-integrations-End-Device-SDK` root folder. Your folder structure should match the following:

```
|-connectedhomeip
|-IoT-managed-integrations-End-Device-SDK
```

Note

You don't need to recursively clone submodules.

Generate code for devices

Create customized C code for your devices using the Managed Integrations code generation tools. This section describes how to generate code from sample files included with the SDK or from your own specifications. Learn how to use the generation scripts, understand the workflow process, and create code that matches your device requirements.

Topics

- Prerequisites
- Generate code for custom .matter files
- Code generation workflow

Prerequisites

1. Python 3.10 or higher.

2. Start with a .matter file for code generation. The End device SDK provides two sample files in the `codgen/matter_files folder`:

- `custom-air-purifier.matter`

- `aws_camera.matter`

> **Note**
>
> These sample files generate code for demo application clusters.

Generate code

Run this command to generate code in the out folder:

```
bash ./gen-data-model-api.sh
```

Generate code for custom .matter files

To generate the code for a specific `.matter` file or provide your own `.matter` file, perform the following tasks.

To generate the code for custom .matter files

1. Prepare your .matter file
2. Run the generation command:

```
./codegen.sh [--format] configs/dm_basic.json path-to-matter-file output-directory
```

(Optional) To generate code with custom schema

1. Prepare your custom schema in `JSON` format
2. Run the generation command:

```
./codegen.sh [--format] configs/dm_basic.json path-to-matter-file output-directory
 --custom-schemas-dir path-to-custom-schema-directory
```

The above commands uses several components to transform your `.matter` file into `C` code:

- `codegen.py` from the **ConnectedHomeIP** project
- Python plugin located at `codegen/py_scripts/iotmi_data_model.py`

- Jinja2 templates from the `codegen/py_scripts/templates` folder

The plugin defines the variables to pass to the Jinja2 templates, which are then used to generate the final C code output. Adding the `--format` flag applies the Clang format to the generated code.

Code generation workflow

The code generation process organizes your .matter file data structures using utility functions and topological sorting through `topsort.py`. This ensures proper ordering of data types and their dependencies.

The script then combines your .matter file specifications with Python plugin processing to extract and format the necessary information. Finally, it applies Jinja2 template formatting to create the final C code output.

This workflow ensures that your device-specific requirements from the .matter file are accurately translated into functional C code that integrates with the Managed Integrations system.

Low level C-Function APIs

Integrate your device-specific code with Managed Integrations using the provided low level C-Function APIs. This section describes the API operations available for each cluster in the AWS data model for efficient device to cloud interactions. Learn how to implement callback functions, emit events, notify attribute changes, and register clusters for your device endpoints.

Key API components include:

1. Callback function pointer structures for attributes and commands
2. Event emission functions
3. Attribute change notification functions
4. Cluster registration functions

By implementing these APIs, you create a bridge between your device's physical operations and the Managed Integrations cloud features, ensuring seamless communication and control.

The following section illustrates the OnOff cluster API.

OnOff cluster API

The OnOff.xml cluster supports these attributes and commands: .

- Attributes:
  - `OnOff (boolean)`
  - `GlobalSceneControl (boolean)`
  - `OnTime (int16u)`
  - `OffWaitTime (int16u)`
  - `StartUpOnOff (StartUpOnOffEnum)`
- Commands:
  - `Off : () -> Status`
  - `On : () -> Status`
  - `Toggle : () -> Status`
  - `OffWithEffect : (EffectIdentifier: EffectIdentifierEnum, EffectVariant: enum8) -> Status`
  - `OnWithRecallGlobalScene : () -> Status`
  - `OnWithTimedOff : (OnOffControl: OnOffControlBitmap, OnTime: int16u, OffWaitTime: int16u) -> Status`

  For each command, we provide the 1:1 mapped function pointer that you can use to hook your implementation.

All the callbacks for attributes and commands are defined within a C struct named after the cluster.

Example C struct

```
struct iotmiDev_clusterOnOff
{
  /*
    - Each attribute has a getter callback if it's readable

    - Each attribute has a setter callback if it's writable

    - The type of `value` are derived according to the data type of
      the attribute.
```

```
    - `user` is the pointer passed during an endpoint setup

    - The callback should return iotmiDev_DMStatus to report success or not.

    - For unsupported attributes, just leave them as NULL.
   */
  iotmiDev_DMStatus (*getOnTime)(uint16_t *value, void *user);
  iotmiDev_DMStatus (*setOnTime)(uint16_t value, void *user);
  /*
    - Each command has a command callback

    - If a command takes parameters, the parameters will be defined in a struct
      such as `iotmiDev_OnOff_OnWithTimedOffRequest` below.

    - `user` is the pointer passed during an endpoint setup

    - The callback should return iotmiDev_DMStatus to report success or not.

    - For unsupported commands, just leave them as NULL.
   */
  iotmiDev_DMStatus (*cmdOff)(void *user);
  iotmiDev_DMStatus (*cmdOnWithTimedOff)(const iotmiDev_OnOff_OnWithTimedOffRequest
 *request, void *user);
};
```

In addition to the C struct, attribute change reporting functions are defined for all attributes.

```
/* Each attribute has a report function for the customer to report
   an attribute change. An attribute report function is thread-safe.
   */
void iotmiDev_OnOff_OnTime_report_attr(struct iotmiDev_Endpoint *endpoint, uint16_t
 newValue, bool immediate);
```

Event reporting functions are defined for all cluster-specific events. Since the OnOff cluster does not define any events, below is an example from the `CameraAvStreamManagement` cluster.

```
/* Each event has a report function for the customer to report
   an event. An event report function is thread-safe.
   The iotmiDev_CameraAvStreamManagement_VideoStreamChangedEvent struct is
   derived from the event definition in the cluster.
 */
```

```
void iotmiDev_CameraAvStreamManagement_VideoStreamChanged_report_event(struct
 iotmiDev_Endpoint *endpoint, const
 iotmiDev_CameraAvStreamManagement_VideoStreamChangedEvent *event, bool immediate);
```

Each cluster also has a register function.

```
iotmiDev_DMStatus iotmiDev_OnOffRegisterCluster(struct iotmiDev_Endpoint *endpoint,
 const struct iotmiDev_clusterOnOff *cluster, void *user);
```

The user pointer passed to the register function will be passed to the callback functions.

Feature and device interactions in Managed Integrations

This section describes the role of the C-Function implementation and the interaction between the device and the Managed Integrations device feature.

Topics

- Handling remote commands
- Handling unsolicited events

Handling remote commands

Remote commands are handled by the interaction between the End device SDK and the feature. The following actions describe an example of how you can turn on a light bulb using this interaction.

MQTT client receives payload and passes to Data Model Handler

When you send a remote command, the MQTT client receives the message from Managed Integrations in JSON format. It then passes the payload to the data model handler. For example, say you want to use Managed Integrations to turn on a light bulb. The light bulb has an endpoint#1 that supports the OnOff cluster. In this case, when you send the command to turn on the light bulb, Managed Integrations sends a request over MQTT to the device, which says that it wants to invoke the On command on endpoint#1.

Data Model Handler checks for callback functions and invokes them

The Data Model Handler parses the JSON request. If the request contains properties or actions, the Data Model Handler finds the endpoints and sequentially invokes the corresponding

callback functions. For example, in the case of the light bulb, when the Data Model Handler receives the MQTT message, it checks whether the callback function corresponding to the On command defined in the OnOff cluster is registered on endpoint#1.

Handler and C-Function implementation execute the command

The Data Model Handler calls the appropriate callback functions that it found and invokes them. The C-Function implementation then calls the corresponding hardware functions to control the physical hardware and returns the execution result. For example, in the case of the light bulb, the Data Model Handler calls the callback function and stores the execution result. The callback function then turns on the light bulb as a result.

Data Model Handler returns execution result

Once all callback functions have been called, the Data Model Handler combines all results. It then packs the response in JSON format and publishes the result to the Managed Integrations cloud using the MQTT client. In the case of the light bulb, the MQTT message in the response will contain the result that the light bulb was turned on by the callback function.

Handling unsolicited events

Unsolicited events are also handled by the interaction between the End device SDK and the feature. The following actions describe how.

Device sends notification to Data Model Handler

When a property change or event occurs, such as when a physical button has been pushed on the device, the C-Function implementation generates an unsolicited event notification and calls the corresponding notify function to send the notification to the Data Model Handler.

Data Model Handler translates notification

The Data Model Handler handles the notification received and translates it to the AWS data model.

Data Model Handler publishes notification to the Cloud

The Data Model Handler then publishes an unsolicited event to the Managed Integrations cloud using the MQTT client.

Get started with End device SDK

Follow these steps to run the End device SDK on a Linux device. This section guides you through environment setup, network configuration, hardware function implementation, and endpoint configuration.

> **Important**
>
> The demonstration applications in the `examples` directory and their Platform Abstraction Layer (PAL) implementation in `platform/posix` are for reference only. Do not use these in production environments.

Review each step of the following procedure carefully to ensure proper device integration with Managed Integrations.

Integrate the End device SDK

1. **Setup Amazon EC2 instance**

   Sign in to the AWS Management Console and launch an Amazon EC2 instance using an Amazon Linux AMI. See Get started with Amazon EC2 in the *Amazon Elastic Container Registry User Guide*.

2. **Set up the build environment**

   Build the code on Amazon Linux 2023/x86_64 as your development host. Install the necessary build dependencies:

   ```
   dnf install make gcc gcc-c++ cmake
   ```

3. **(Optional) Set up the network**

   End device SDK is best used with physical hardware. If using Amazon EC2, do not follow this step.

   If you are not using Amazon EC2 before using the sample application, initialize the network and connect your device to an available Wi-Fi network. Complete the network setup before device provisioning:

```
/* Provisioning the device PKCS11 with claim credential. */
status = deviceCredentialProvisioning();
```

4. **Configure provisioning parameters**

> **Note**
>
> Follow Provisionee to get the claim certificate and private key before proceeding further.

Modify the configuration file `example/project_name/device_config.sh` with the following provisioning parameters:

Provisioning parameters

Macro parameters	Description	How to obtain this information
`IOTMI_ROOT_CA_PATH`	The root CA certificate file.	You can download this file from the Download the Amazon Root CA certificate section in the *AWS IoT Core developer guide*.
`IOTMI_CLAIM_CERTIFICATE_PATH`	The path to the claim certificate file.	To obtain the claim certificate and private key, create a provisioning profile using the CreateProvisioningProfile API. For instructions, see Create a provisioning profile.
`IOTMI_CLAIM_PRIVATE_KEY_PATH`	The path to the claim private key file.	
`IOTMI_MANAGEDINTEGRATIONS_ENDPOINT`	Endpoint URL for Managed Integrations.	To obtain the Managed Integrations endpoint, use the RegisterCustomEndpoint API. For instructions, see Register a custom endpoint.

Macro parameters	Description	How to obtain this information
IOTMI_MANAGEDINTEGRATIONS_ENDPOINT_PORT	The port number for the Managed Integrations endpoint	By default, the port 8883 is used for MQTT publish and subscribe operations. Port 443 is set for Application Layer Protocol Negotiation (ALPN) TLS extension that devices use.

5. **Build and run the demo applications**

   This section demonstrates two Linux demo applications: a simple security camera and an air purifier, both using CMake as the build system.

   a. Simple security camera application

      To build and run the application, execute these commands:

```
>cd <path-to-code-drop>
# If you didn't generate cluster code earlier
>(cd codegen && poetry run poetry install --no-root && ./gen-data-model-api.sh)
>mkdir build
>cd build
>cmake ..
>cmake –build .
>./examples/iotmi_device_sample_camera/iotmi_device_sample_camera
```

      This demo implements low-level C-Functions for a simulated camera with RTC Session Controller and Recording clusters. Complete the flow mentioned in Provisionee workflow before running.

      Sample output of the demo application:

```
[2406832727][MAIN][INFO] ======= Device initialization and WIFI provisioning
 =======
[2406832728][MAIN][INFO] fleetProvisioningTemplateName: XXXXXXXXXXX
[2406832728][MAIN][INFO] managedintegrationsEndpoint: XXXXXXXXX.account-prefix-
ats.iot.region.amazonaws.com
[2406832728][MAIN][INFO] pDeviceSerialNumber: XXXXXXXXXXX
[2406832728][MAIN][INFO] universalProductCode: XXXXXXXXXXX
[2406832728][MAIN][INFO] rootCertificatePath: XXXXXXXXX
```

```
[2406832728][MAIN][INFO] pClaimCertificatePath: XXXXXXXX
[2406832728][MAIN][INFO] pClaimKeyPath: XXXXXXXXXXXXXXXX
[2406832728][MAIN][INFO] deviceInfo.serialNumber XXXXXXXXXXX
[2406832728][MAIN][INFO] deviceInfo.universalProductCode XXXXXXXXXXXXXX
[2406832728][PKCS11][INFO] PKCS #11 successfully initialized.
[2406832728][MAIN][INFO] ============= Start certificate provisioning
 =============
[2406832728][PKCS11][INFO] ======== Loading Root CA and claim credentials
 through PKCS#11 interface ========
[2406832728][PKCS11][INFO] Writing certificate into label "Root Cert".
[2406832728][PKCS11][INFO] Creating a 0x1 type object.
[2406832728][PKCS11][INFO] Writing certificate into label "Claim Cert".
[2406832728][PKCS11][INFO] Creating a 0x1 type object.
[2406832728][PKCS11][INFO] Creating a 0x3 type object.
[2406832728][MAIN][INFO] ======== Fleet-provisioning-by-Claim ========
[2025-01-02 01:43:11.404995144][iotmi_device_sdkLog][INFO] [2406832728]
[MQTT_AGENT][INFO]
[2025-01-02 01:43:11.405106991][iotmi_device_sdkLog][INFO] Establishing a TLS
 session to XXXXXXXXXXXXXXX.account-prefix-ats.iot.region.amazonaws.com
[2025-01-02 01:43:11.405119166][iotmi_device_sdkLog][INFO]
[2025-01-02 01:43:11.844812513][iotmi_device_sdkLog][INFO] [2406833168]
[MQTT_AGENT][INFO]
[2025-01-02 01:43:11.844842576][iotmi_device_sdkLog][INFO] TLS session
 connected
[2025-01-02 01:43:11.844852105][iotmi_device_sdkLog][INFO]
[2025-01-02 01:43:12.296421687][iotmi_device_sdkLog][INFO] [2406833620]
[MQTT_AGENT][INFO]
[2025-01-02 01:43:12.296449663][iotmi_device_sdkLog][INFO] Session present: 0.
[2025-01-02 01:43:12.296458997][iotmi_device_sdkLog][INFO]
[2025-01-02 01:43:12.296467793][iotmi_device_sdkLog][INFO] [2406833620]
[MQTT_AGENT][INFO]
[2025-01-02 01:43:12.296476275][iotmi_device_sdkLog][INFO] MQTT connect with
 clean session.
[2025-01-02 01:43:12.296484350][iotmi_device_sdkLog][INFO]
[2025-01-02 01:43:13.171056119][iotmi_device_sdkLog][INFO] [2406834494]
[FLEET_PROVISIONING][INFO]
[2025-01-02 01:43:13.171082442][iotmi_device_sdkLog][INFO] Received accepted
 response from Fleet Provisioning CreateKeysAndCertificate API.
[2025-01-02 01:43:13.171092740][iotmi_device_sdkLog][INFO]
[2025-01-02 01:43:13.171122834][iotmi_device_sdkLog][INFO] [2406834494]
[FLEET_PROVISIONING][INFO]
[2025-01-02 01:43:13.171132400][iotmi_device_sdkLog][INFO] Received privatekey
 and certificate with Id: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[2025-01-02 01:43:13.171141107][iotmi_device_sdkLog][INFO]
```

```
[2406834494][PKCS11][INFO] Creating a 0x3 type object.
[2406834494][PKCS11][INFO] Writing certificate into label "Device Cert".
[2406834494][PKCS11][INFO] Creating a 0x1 type object.
[2025-01-02 01:43:18.584615126][iotmi_device_sdkLog][INFO] [2406839908]
[FLEET_PROVISIONING][INFO]
[2025-01-02 01:43:18.584662031][iotmi_device_sdkLog][INFO] Received accepted
 response from Fleet Provisioning RegisterThing API.
[2025-01-02 01:43:18.584671912][iotmi_device_sdkLog][INFO]
[2025-01-02 01:43:19.100030237][iotmi_device_sdkLog][INFO] [2406840423]
[FLEET_PROVISIONING][INFO]
[2025-01-02 01:43:19.100061720][iotmi_device_sdkLog][INFO] Fleet-provisioning
 iteration 1 is successful.
[2025-01-02 01:43:19.100072401][iotmi_device_sdkLog][INFO]
[2406840423][MQTT][ERROR] MQTT Connection Disconnected Successfully
[2025-01-02 01:43:19.216938181][iotmi_device_sdkLog][INFO] [2406840540]
[MQTT_AGENT][INFO]
[2025-01-02 01:43:19.216963713][iotmi_device_sdkLog][INFO] MQTT agent thread
 leaves thread loop for iotmiDev_MQTTAgentStop.
[2025-01-02 01:43:19.216973740][iotmi_device_sdkLog][INFO]
[2406840540][MAIN][INFO] iotmiDev_MQTTAgentStop is called to break thread loop
 function.
[2406840540][MAIN][INFO] Successfully provision the device.
[2406840540][MAIN][INFO] Client ID :
 XXXXXXXXXXXXXXXXXXX_XXXXXXXXXXXXXXXXXXXXXX
[2406840540][MAIN][INFO] Managed thing ID : XXXXXXXXXXXXXXXXXXXXX
[2406840540][MAIN][INFO] ======================== application loop
 =================
[2025-01-02 01:43:19.217094828][iotmi_device_sdkLog][INFO] [2406840540]
[MQTT_AGENT][INFO]
[2025-01-02 01:43:19.217124600][iotmi_device_sdkLog][INFO] Establishing a TLS
 session to XXXXXXXXX.account-prefix-ats.iot.region.amazonaws.com:8883
[2025-01-02 01:43:19.217138724][iotmi_device_sdkLog][INFO]
[2406840540][Cluster OnOff][INFO] exampleOnOffInitCluster() for endpoint#1
[2406840540][MAIN][INFO] Press Ctrl+C when you finish testing...
[2406840540][Cluster ActivatedCarbonFilterMonitoring][INFO]
 exampleActivatedCarbonFilterMonitoringInitCluster() for endpoint#1
[2406840540][Cluster AirQuality][INFO] exampleAirQualityInitCluster() for
 endpoint#1
[2406840540][Cluster CarbonDioxideConcentrationMeasurement][INFO]
 exampleCarbonDioxideConcentrationMeasurementInitCluster() for endpoint#1
[2406840540][Cluster FanControl][INFO] exampleFanControlInitCluster() for
 endpoint#1
[2406840540][Cluster HepaFilterMonitoring][INFO]
 exampleHepaFilterMonitoringInitCluster() for endpoint#1
```

```
[2406840540][Cluster Pm1ConcentrationMeasurement][INFO]
 examplePm1ConcentrationMeasurementInitCluster() for endpoint#1
[2406840540][Cluster Pm25ConcentrationMeasurement][INFO]
 examplePm25ConcentrationMeasurementInitCluster() for endpoint#1
[2406840540][Cluster TotalVolatileOrganicCompoundsConcentrationMeasurement]
[INFO]
 exampleTotalVolatileOrganicCompoundsConcentrationMeasurementInitCluster() for
 endpoint#1
[2025-01-02 01:43:19.648185488][iotmi_device_sdkLog][INFO] [2406840971]
[MQTT_AGENT][INFO]
[2025-01-02 01:43:19.648211988][iotmi_device_sdkLog][INFO] TLS session
 connected
[2025-01-02 01:43:19.648225583][iotmi_device_sdkLog][INFO]

[2025-01-02 01:43:19.938281231][iotmi_device_sdkLog][INFO] [2406841261]
[MQTT_AGENT][INFO]
[2025-01-02 01:43:19.938304799][iotmi_device_sdkLog][INFO] Session present: 0.
[2025-01-02 01:43:19.938317404][iotmi_device_sdkLog][INFO]
```

b. Simple air purifier application

To build and run the application, run the following commands:

```
>cd <path-to-code-drop>
# If you didn't generate cluster code earlier
>(cd codegen && poetry run poetry install --no-root && ./gen-data-model-api.sh)
>mkdir build
>cd build
>cmake ..
>cmake --build .
>./examples/iotmi_device_dm_air_purifier/iotmi_device_dm_air_purifier_demo
```

This demo implements low-level C-Functions for a simulated air purifier with 2 endpoints and the following supported clusters:

Supported clusters for air purifier endpoint

Endpoint	Clusters
Endpoint #1: Air Purifier	OnOff
	Fan Control

Endpoint	Clusters
	HEPA Filter Monitoring
	Activated Carbon Filter Monitoring
Endpoint #2: Air Quality Sensor	Air Quality
	Carbon Dioxide Concentration Measurement
	Formaldehyde Concentration Measurement
	Pm25 Concentration Measurement
	Pm1 Concentration Measurement
	Total Volatile Organic Compounds Concentration Measurement

The output is similar to the camera demo application, with different supported clusters.

6. **Next steps:**

The Managed Integrations End device SDK and demo applications are now running on your Amazon EC2 instances. This allows you to develop and test your applications on your own physical hardware. With this setup, you can leverage the Managed Integrations service to control your AWS IoT devices.

a. **Develop hardware callback functions**

Before implementing the hardware callback functions, understand how the API works. This example uses the On/Off cluster and OnOff attribute to control a device function. For API details, see Low level C-Function APIs.

```
struct DeviceState
{
  struct iotmiDev_Agent *agent;
  struct iotmiDev_Endpoint *endpointLight;
  /* This simulates the HW state of OnOff */
  bool hwState;
};
```

```
/* This implementation for OnOff getter just reads
   the state from the DeviceState */
iotmiDev_DMStatus exampleGetOnOff(bool *value, void *user)
{
   struct DeviceState *state = (struct DeviceState *)(user);
  *value = state->hwState;
  return iotmiDev_DMStatusOk;
}
```

b. **Set up endpoints and hook hardware callback functions**

After implementing the functions, create endpoints and register your callbacks. Complete the following tasks:

i. Create a device agent.

A. Create a device agent using `iotmiDev_Agent_new()` before you invoke any other SDK functions.

B. At minimum, your configuration must include the **thingId** and **clientId** parameters.

C. Use the `iotmiDev_Agent_initDefaultConfig()` function to set reasonable default values for parameters such as queue sizes and maximum endpoints.

D. When you finish using resources, free them with the `iotmiDev_Agent_free()` function. This prevents memory leaks and ensures proper resource management in your application.

ii. Fill callback function pointers for each cluster structure that you want to support.

iii. Set up endpoints and register your supported clusters.

Create endpoints with `iotmiDev_Agent_addEndpoint()`, which requires:

A. A unique endpoint ID.

B. A descriptive endpoint name

C. One or more device types that conform to AWS data model definitions.

D. After creating endpoints, register clusters using the appropriate cluster-specific registration functions.

E. Each cluster registration requires callback functions for attributes and commands. The system passes your user context pointer to callbacks to maintain state between calls.

```
struct DeviceState
{
    struct iotmiDev_Agent * agent;
    struct iotmiDev_Endpoint *endpoint1;

    /* OnOff cluster states*/
    bool hwState;
};


/* This implementation for OnOff getter just reads
   the state from the DeviceState */
iotmiDev_DMStatus exampleGetOnOff( bool * value, void * user )
{
    struct DeviceState * state = ( struct DeviceState * ) ( user );
    *value = state->hwState;
    printf( "%s(): state->hwState: %d\n", __func__, state->hwState );
    return iotmiDev_DMStatusOk;
}

iotmiDev_DMStatus exampleGetOnTime( uint16_t * value, void * user )
{
    *value = 0;
    printf( "%s(): OnTime is %u\n", __func__, *value );
    return iotmiDev_DMStatusOk;
}

iotmiDev_DMStatus exampleGetStartUpOnOff( iotmiDev_OnOff_StartUpOnOffEnum *
 value, void * user )
{
    *value = iotmiDev_OnOff_StartUpOnOffEnum_Off;
    printf( "%s(): StartUpOnOff is %d\n", __func__, *value );
    return iotmiDev_DMStatusOk;
}

void setupOnOff( struct DeviceState *state )
{
    struct iotmiDev_clusterOnOff clusterOnOff = {
```

```
        .getOnOff = exampleGetOnOff,
        .getOnTime = exampleGetOnTime,
        .getStartUpOnOff = exampleGetStartUpOnOff,
    };
    iotmiDev_OnOffRegisterCluster( state->endpoint1,
                                   &clusterOnOff,
                                   ( void * ) state);
}


/* Here is the sample setting up an endpoint 1 with OnOff
   cluster. Note all error handling code is omitted. */
void setupAgent(struct DeviceState *state)
{
    struct iotmiDev_Agent_Config config = {
        .thingId = IOTMI_DEVICE_MANAGED_THING_ID,
        .clientId = IOTMI_DEVICE_CLIENT_ID,
    };
    iotmiDev_Agent_InitDefaultConfig(&config);

    /* Create a device agent before calling other SDK APIs */
    state->agent = iotmiDev_Agent_new(&config);

    /* Create endpoint#1 */
    state->endpoint1 = iotmiDev_Agent_addEndpoint( state->agent,
                                                   1,
                                                   "Data Model Handler Test
 Device",
                                                   (const char*[])
{ "Camera" },
                                                   1 );
    setupOnOff(state);
}
```

c. **Use the jobs handler to obtain the job document**

i. Initiate a call to your OTA application:

```
static iotmi_JobCurrentStatus_t processOTA( iotmi_JobData_t * pJobData )
{
    iotmi_JobCurrentStatus_t jobCurrentStatus = JobSucceeded;

...
    // This function should create OTA tasks
```

```
    jobCurrentStatus = YOUR_OTA_FUNCTION(iotmi_JobData_t * pJobData);
...

    return jobCurrentStatus;
}
```

ii. Call `iotmi_JobsHandler_start` to initialize the jobs handler.

iii. Call `iotmi_JobsHandler_getJobDocument` to retrieve the job Document from Managed Integrations.

iv. When the Jobs Document is obtained successfully, write your custom OTA operation in the `processOTA` function and return a `JobSucceeded` status.

```
static void prvJobsHandlerThread( void * pParam )
{
    JobsHandlerStatus_t status = JobsHandlerSuccess;
    iotmi_JobData_t jobDocument;
    iotmiDev_DeviceRecord_t * pThreadParams = ( iotmiDev_DeviceRecord_t * )
 pParam;
    iotmi_JobsHandler_config_t config = { .pManagedThingID = pThreadParams-
>pManagedThingID, .jobsQueueSize = 10 };

    status = iotmi_JobsHandler_start( &config );

    if( status != JobsHandlerSuccess )
    {
        LogError( ( "Failed to start Jobs Handler." ) );
        return;
    }

    while( !bExit )
    {
        status = iotmi_JobsHandler_getJobDocument( &jobDocument, 30000 );

        switch( status )
        {
            case JobsHandlerSuccess:
            {
                LogInfo( ( "Job document received." ) );
                LogInfo( ( "Job ID: %.*s", ( int ) jobDocument.jobIdLength,
 jobDocument.pJobId ) );
                LogInfo( ( "Job document: %.*s", ( int )
 jobDocument.jobDocumentLength, jobDocument.pJobDocument ) );
```

```
                /* Process the job document */
                iotmi_JobCurrentStatus_t jobStatus =
processOTA( &jobDocument );

                iotmi_JobsHandler_updateJobStatus( jobDocument.pJobId,
jobDocument.jobIdLength, jobStatus, NULL, 0 );

                iotmiJobsHandler_destroyJobDocument(&jobDocument);

                break;
            }
            case JobsHandlerTimeout:
            {
                LogInfo( ( "No job document available. Polling for job
document." ) );

                iotmi_JobsHandler_pollJobDocument();

                break;
            }
            default:
            {
                LogError( ( "Failed to get job document." ) );
                break;
            }
        }
    }

    while( iotmi_JobsHandler_getJobDocument( &jobDocument, 0 ) ==
JobsHandlerSuccess )
    {
        /* Before stopping the Jobs Handler, process all the remaining
jobs. */

        LogInfo( ( "Job document received before stopping." ) );
        LogInfo( ( "Job ID: %.*s", ( int ) jobDocument.jobIdLength,
jobDocument.pJobId ) );
        LogInfo( ( "Job document: %.*s", ( int )
jobDocument.jobDocumentLength, jobDocument.pJobDocument ) );

        storeJobs( &jobDocument );

        iotmiJobsHandler_destroyJobDocument(&jobDocument);
```

```
    }

    iotmi_JobsHandler_stop();

    LogInfo( ( "Job handler thread end." ) );

}
```

Port the End device SDK to your device

Port the End device SDK to your device platform. Follow these steps to connect your devices with AWS IoT Device Management.

Download and verify the End device SDK

1. Download the latest version of the End device SDK from the managed integrations console.
2. Verify that your platform is in the list of supported platforms in Reference: Supported platforms.

   > **Note**
   >
   > The End device SDK has been tested on the specified platforms. Other platforms might work, but haven't been tested.

3. Extract (unzip) the SDK files to your workspace.
4. Configure your build environment with the following settings:

   - Source file paths
   - Header file directories
   - Required libraries
   - Compiler and linker flags

5. Before you port the Platform Abstraction Layer (PAL), make sure your platform's basic functionalities are initialized. Functionalities include:

   - Operating system tasks
   - Peripherals
   - Network interfaces

- Platform-specific requirements

Port the PAL to your device

1. Create a new directory for your platform-specific implementations in the existing platform directory. For example, if you use FreeRTOS, create a directory at `platform/freertos`.

   Example SDK directory structure

   ```
   ### <SDK_ROOT_FOLDER>
   #   ### CMakeLists.txt
   #   ### LICENSE.txt
   #   ### cmake
   #   ### commonDependencies
   #   ### components
   #   ### docs
   #   ### examples
   #   ### include
   #   ### lib
   #   ### platform
   #   ### test
   #   ### tools
   ```

2. Copy the POSIX reference implementation files (.c and .h) from the posix folder to your new platform directory. These files provide a template for the functions you'll need to implement.

   - Flash memory management for credential storage
   - PKCS#11 implementation
   - Network transport interface
   - Time synchronization
   - System reboot and reset functions
   - Logging mechanisms
   - Device-specific configurations

3. Set up Transport Layer Security (TLS) authentication with MBedTLS.

   - Use the provided POSIX implementation if you already have an MBedTLS version that matches the SDK version on your platform.

- With a different TLS version, you implement the transport hooks for your TLS stack with TCP/IP stack.

4. Compare your platform's MbedTLS configuration with the SDK requirements in `platform/posix/mbedtls/mbedtls_config.h` . Make sure all of the required options are enabled.
5. The SDK relies on the coreMQTT to interact with cloud. Therefore, you must implement a network transport layer that uses the following structure:

```
typedef struct TransportInterface
{
    TransportRecv_t recv;
    TransportSend_t send;
    NetworkContext_t * pNetworkContext;
} TransportInterface_t;
```

   For more information, see Transport interface documentation on the *FreeRTOS* website.
6. (Optional) The SDK uses the PCKS#11 API to handle certificate operations. corePKCS is a non hardware specific PKCS#11 implementation for prototyping. We recommend you use secure cryptoprocessors such as Trusted Platform Module (TPM), Hardware Security Module (HSM), or Secure Element in your production environment:

   - Review the sample PKCS#11 implementation that uses the Linux file system for credential management at `platform/posix/corePKCS11-mbedtls`.
   - Implement the PKCS#11 PAL layer at `commonDependencies/core_pkcs11/corePKCS11/source/include/core_pkcs11.h`.
   - Implement the Linux file system at `platform/posix/corePKCS11-mbedtls/source/iotmi_pal_Pkcs11Operations.c`.
   - Implement the store and load function of your storage type at `platform/include/iotmi_pal_Nvm.h`.
   - Implement the standard file access at `platform/posix/source/iotmi_pal_Nvm.c`.

   For detailed porting instructions, see Porting the corePKCS11 library in the *FreeRTOS User Guide*.
7. Add the SDK static libraries to your build environment:

   - Set up the library paths to resolve any linker issues or symbol conflicts

- Verify all dependencies are properly linked

Test your port

You can use the existing example application to test your port. The compilation must complete without any errors or warnings.

> **Note**
>
> We recommend that you start with the simplest possible multitasking application. The example application provides a multitasking equivalent.

1. Find the example application in `examples/[device_type_sample]`.
2. Convert the `main.c` file to your project, and add an entry to call the existing main() function.
3. Verify that you can compile the demo application successfully.

Technical reference

Topics

- Reference: Supported platforms
- Reference: Technical requirements
- Reference: Common API

Reference: Supported platforms

The following table displays the supported platforms for the SDK.

Supported platforms

Platform	Architecture	Operating system
Linux x86_64	x86_64	Linux
Ambarella	Armv8 (AArch64)	Linux
AmebaD	Armv8-M 32 bit	FreeRTOS

Platform	Architecture	Operating system
ESP32S3	Xtensa LX7 32 bit	FreeRTOS

Reference: Technical requirements

The following table shows the technical requirements for the SDK, including the RAM space. The End device SDK itself requires about 5 to 10 MB of ROM space when using the same configuration.

RAM space

SDK and components	Space requirements (bytes used)
End device SDK itself	180 KB
Default MQTT Agent command queue	480 bytes (can be configured)
Default MQTT Agent incoming queue	320 bytes (can be configured)

Reference: Common API

This section is a list of API operations that are not specific to a cluster.

```
/* return code for data model related API */
enum iotmiDev_DMStatus
{
  /* The operation succeeded */
  iotmiDev_DMStatusOk = 0,
  /* The operation failed without additional information */
  iotmiDev_DMStatusFail = 1,
  /* The operation has not been implemented yet. */
  iotmiDev_DMStatusNotImplement = 2,
  /* The operation is to create a resource, but the resource already exists. */
  iotmiDev_DMStatusExist = 3,
}

/* The opaque type to represent a instance of device agent. */
struct iotmiDev_Agent;

/* The opaque type to represent an endpoint. */
```

```
struct iotmiDev_Endpoint;

/* A device agent should be created before calling other API */
struct iotmiDev_Agent* iotmiDev_create_agent();

/* Destroy the agent and free all occupied resources */
void iotmiDev_destroy_agent(struct iotmiDev_Agent *agent);

/* Add an endpoint, which starts with empty capabilities */
struct iotmiDev_Endpoint* iotmiDev_addEndpoint(struct iotmiDev_Agent *handle, uint16
 id, const char *name);

/* Test all clusters registered within an endpoint.
   Note: this API might exist only for early drop. */
void iotmiDev_testEndpoint(struct iotmiDev_Endpoint *endpoint);
```

Security in managed integrations for AWS IoT Device Management

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from data centers and network architectures that are built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS Compliance Programs. To learn about the compliance programs that apply to Managed Integrations, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Managed Integrations. The following topics show you how to configure Managed Integrations to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your Managed Integrations resources.

Topics

- Data protection in Managed Integrations
- Identity and access management for Managed Integrations
- Use AWS Secrets Manager for data protection for C2C workflows
- Compliance validation for Managed Integrations
- Use Managed Integrations with interface VPC endpoints
- Connect to managed integrations for AWS IoT Device Management FIPS endpoints

Data protection in Managed Integrations

The AWS shared responsibility model applies to data protection in Managed Integrations for AWS IoT Device Management. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see Data Privacy FAQ. For information about data protection in Europe, see the General Data Protection Regulation (GDPR) Center.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-3.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with managed integrations for AWS IoT Device Management or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Data encryption at rest for Managed Integrations

Managed Integrations for AWS IoT Device Management encrypts sensitive customer data at rest by default using encryption keys.

There are two types of encryption keys that are used to protect sensitive data for Managed Integrations customers:

Customer managed keys (CMK)

Managed Integrations supports the use of symmetric customer managed key that you can create, own, and manage. You have full control over these KMS keys, including establishing and maintaining their key policies, IAM policies, and grants, enabling and disabling them, rotating their cryptographic material, adding tags, creating aliases that refer to the KMS keys, and scheduling the KMS keys for deletion.

AWS owned keys

Managed Integrations uses these keys by default to automatically encrypt sensitive customer data. You can't view, manage, or audit their use. You don't have to take any action or change any programs to protect the keys that encrypt your data. Encryption of data at rest by default helps reduce the operational overhead and complexity involved in protecting sensitive data. At the same time, it enables you to build secure applications that meet strict encryption compliance and regulatory requirements.

The default encryption key used is AWS owned keys. Alternatively, the optional API to update your encryption key is `PutDefaultEncryptionConfiguration`.

For more information on the types of AWS KMS encryption keys, see AWS KMS keys.

AWS KMS usage for Managed Integrations

Managed Integrations encrypts and decrypts all customer data using envelope encryption. This type of encryption will take your plaintext data and encrypt it with a data key. Next, an encryption key called a wrapping key will encrypt the original data key used to encrypt your plaintext data. In envelope encryption, additional wrapping keys can be used to encrypt existing wrapping keys that are closer in degrees of separation from the original data key. Since the original data key is encrypted by a wrapping key stored separately, you can store the original data key and encrypted plaintext data in the same location. A keyring is used to generate, encrypt, and decrypt data keys in addition to which wrapping key is used to encrypt and decrypt the data key.

> **Note**
>
> The AWS Database Encryption SDK provides envelope encryption for your client-side encryption implementation. For more information on the AWS Database Encryption SDK, see What is the AWS Database Encryption SDK?

For more information on envelope encryption, data keys, wrapping keys, and keyrings, see Envelope encryption, Data key, Wrapping key, and Keyrings.

Managed Integrations requires the services to use your customer managed key for the following internal operations:

- Send `DescribeKey` requests to AWS KMS to verify that the symmetric customer managed key ID provided when doing the rotation of the data keys.
- Send `GenerateDataKeyWithoutPlaintext` requests to AWS KMS to generate data keys encrypted by your customer managed key.
- Send `ReEncrypt*` requests to AWS KMS to reencrypt data keys by your customer managed key.
- Send `Decrypt` requests to AWS KMS to decrypt data by your customer managed key.

Types of data encrypted using encryption keys

Managed Integrations uses encryption keys to encrypt multiple types of data stored at rest. The following list outlines types of data encrypted at rest using encryption keys:

- Cloud--to-cloud (C2C) connector events such as device discovery and device status update.
- Creation of a *Managed Thing* that represents the physical device and a device profile containing the capabilities for a specific device type. For more information on a device and device profile, see Device and Device.
- Managed Integrations notifications on various aspects of your device implementation. For more information on Managed Integrations notifications, see Set up Managed Integrations notifications.
- Personally Identifiable Information (PII) of an end-user such as device authentication material, device serial number, end-user's name, device identifier, and device Amazon Resource Name (arn).

How Managed Integrations uses key policies in AWS KMS

For branch key rotation and asynchronous calls, Managed Integrations requires a key policy to use your encryption key. A key policy is used for the following reasons:

- Programmatically authorize the use of an encryption key to other AWS principals.

For an example of a key policy used to manage access to your encryption key in Managed Integrations, see Create an encryption key

> **Note**
>
> For an AWS owned key, a key policy is not required as the AWS owned key is owned by AWS and you can't view, manage, or use it. Managed Integrations uses the AWS owned key by default to automatically encrypt your sensitive customer data.

In addition to using key policies for managing your encryption configuration with AWS KMS keys, Managed Integrations uses IAM policies. For more information on IAM policies, see Policies and permissions in AWS Identity and Access Management.

Create an encryption key

You can create an encryption key by using the AWS Management Console or the AWS KMS APIs.

To create an encryption key

Follow the steps for Creating a KMS key in the AWS Key Management Service Developer Guide.

Key policy

A key policy statement controls access to an AWS KMS key. Each AWS KMS key will contain only one key policy. That key policy determines which AWS principals may use the key and how they may use it. For more information on managing access and usage of AWS KMS keys using key policy statements, see Managing access using policies.

The following is an example of a key policy statement you can use for managing access and usage to AWS KMS keys stored in your AWS account for Managed Integrations:

```
{
   "Statement" : [
```

```
    {
      "Sid" : "Allow access to principals authorized to use managed integrations",
      "Effect" : "Allow",
      "Principal" : {
        //Note: Both role and user are acceptable.
        "AWS": "arn:aws:iam::111122223333:user/username",
        "AWS": "arn:aws:iam::111122223333:role/roleName"
      },
      "Action" : [
        "kms:GenerateDataKeyWithoutPlaintext",
        "kms:Decrypt",
        "kms:ReEncrypt*"
      ],
      "Resource" : "arn:aws:kms:region:111122223333:key/key_ID",
      "Condition" : {
        "StringEquals" : {
          "kms:ViaService" : "iotmanagedintegrations.amazonaws.com"
        },
        "ForAnyValue:StringEquals": {
          "kms:EncryptionContext:aws-crypto-ec:iotmanagedintegrations": "111122223333"
        },
        "ArnLike": {
          "aws:SourceArn": [
            "arn:aws:iotmanagedintegrations:<region>:<accountId>:managed-thing/
<managedThingId>",
            "arn:aws:iotmanagedintegrations:<region>:<accountId>:credential-locker/
<credentialLockerId>",
            "arn:aws:iotmanagedintegrations:<region>:<accountId>:provisioning-profile/
<provisioningProfileId>",
            "arn:aws:iotmanagedintegrations:<region>:<accountId>:ota-task/<otaTaskId>"
          ]
        }
      }
    },
    {
      "Sid" : "Allow access to principals authorized to use managed integrations for
 async flow",
      "Effect" : "Allow",
      "Principal" : {
        "Service": "iotmanagedintegrations.amazonaws.com"
      },
      "Action" : [
        "kms:GenerateDataKeyWithoutPlaintext",
        "kms:Decrypt",
```

```
        "kms:ReEncrypt*"
      ],
      "Resource" : "arn:aws:kms:region:111122223333:key/key_ID",
      "Condition" : {
        "ForAnyValue:StringEquals": {
          "kms:EncryptionContext:aws-crypto-ec:iotmanagedintegrations": "111122223333"
        },
        "ArnLike": {
          "aws:SourceArn": [
            "arn:aws:iotmanagedintegrations:<region>:<accountId>:managed-thing/
<managedThingId>",
            "arn:aws:iotmanagedintegrations:<region>:<accountId>:credential-locker/
<credentialLockerId>",
            "arn:aws:iotmanagedintegrations:<region>:<accountId>:provisioning-profile/
<provisioningProfileId>",
            "arn:aws:iotmanagedintegrations:<region>:<accountId>:ota-task/<otaTaskId>"
          ]
        }
      }
    },
    {
      "Sid" : "Allow access to principals authorized to use managed integrations for
 describe key",
      "Effect" : "Allow",
      "Principal" : {
        "AWS": "arn:aws:iam::111122223333:user/username"
      },
      "Action" : [
        "kms:DescribeKey",
      ],
      "Resource" : "arn:aws:kms:region:111122223333:key/key_ID",
      "Condition" : {
        "StringEquals" : {
          "kms:ViaService" : "iotmanagedintegrations.amazonaws.com"
        }
      }
    },
    {
      "Sid": "Allow access for key administrators",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::111122223333:root"
       },
      "Action" : [
```

```
        "kms:*"
      ],
      "Resource": "*"
    }
  ]
}
```

For more information on key stores, see Key stores.

Updating encryption configuration

The ability to seamlessly update your encryption configuration is critical to managing your data encryption implementation for Managed Integrations. When you initially onboard with Managed Integrations, you will be prompted to select your encryption configuration. Your options will be either the default AWS owned keys or create your own AWS KMS key.

AWS Management Console

To update your encryption configuration in the AWS Management Console, open the AWS IoT service homepage and then navigate to **Managed Integration for Unified Control**>**Settings**>**Encryption**. In the **Encryption settings** window, you can update your encryption configuration by selecting a new AWS KMS key for additional encryption protection. Choose **Customize encryption settings (advanced)** to select an existing AWS KMS key or you can chose **Create an AWS KMS key** to create your own customer managed key.

API Commands

There are two APIs used for managing your encryption configuration of AWS KMS keys in Managed Integrations: `PutDefaultEncryptionConfiuration` and `GetDefaultEncryptionConfiguration`.

To update the default encryption configuration, call `PutDefaultEncryptionConfiuration`. For more information on `PutDefaultEncryptionConfiuration`, see PutDefaultEncryptionConfiuration.

To view the default encryption configuration, call `GetDefaultEncryptionConfiguration`. For more information on `GetDefaultEncryptionConfiguration`, see GetDefaultEncryptionConfiguration.

Identity and access management for Managed Integrations

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use Managed Integrations resources. IAM is an AWS service that you can use with no additional charge.

Topics

- Audience
- Authenticating with identities
- Managing access using policies
- AWS managed policies for Managed Integrations
- How Managed Integrations works with IAM
- Identity-based policy examples for Managed Integrations
- Troubleshooting Managed Integrations identity and access
- Using service-linked roles for managed integrations

Audience

How you use AWS Identity and Access Management (IAM) differs based on your role:

- **Service user** - request permissions from your administrator if you cannot access features (see Troubleshooting Managed Integrations identity and access)
- **Service administrator** - determine user access and submit permission requests (see How Managed Integrations works with IAM)
- **IAM administrator** - write policies to manage access (see Identity-based policy examples for Managed Integrations)

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be authenticated as the AWS account root user, an IAM user, or by assuming an IAM role.

You can sign in as a federated identity using credentials from an identity source like AWS IAM Identity Center (IAM Identity Center), single sign-on authentication, or Google/Facebook

credentials. For more information about signing in, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

For programmatic access, AWS provides an SDK and CLI to cryptographically sign requests. For more information, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity called the AWS account *root user* that has complete access to all AWS services and resources. We strongly recommend that you don't use the root user for everyday tasks. For tasks that require root user credentials, see Tasks that require root user credentials in the *IAM User Guide*.

Federated identity

As a best practice, require human users to use federation with an identity provider to access AWS services using temporary credentials.

A *federated identity* is a user from your enterprise directory, web identity provider, or Directory Service that accesses AWS services using credentials from an identity source. Federated identities assume roles that provide temporary credentials.

For centralized access management, we recommend AWS IAM Identity Center. For more information, see What is IAM Identity Center? in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An *IAM user* is an identity with specific permissions for a single person or application. We recommend using temporary credentials instead of IAM users with long-term credentials. For more information, see Require human users to use federation with an identity provider to access AWS using temporary credentials in the *IAM User Guide*.

An *IAM group* specifies a collection of IAM users and makes permissions easier to manage for large sets of users. For more information, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity with specific permissions that provides temporary credentials. You can assume a role by switching from a user to an IAM role (console) or by calling an AWS CLI or AWS API operation. For more information, see Methods to assume a role in the *IAM User Guide*.

IAM roles are useful for federated user access, temporary IAM user permissions, cross-account access, cross-service access, and applications running on Amazon EC2. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy defines permissions when associated with an identity or resource. AWS evaluates these policies when a principal makes a request. Most policies are stored in AWS as JSON documents. For more information about JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Using policies, administrators specify who has access to what by defining which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. An IAM administrator creates IAM policies and adds them to roles, which users can then assume. IAM policies define permissions regardless of the method used to perform the operation.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you attach to an identity (user, group, or role). These policies control what actions identities can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be *inline policies* (embedded directly into a single identity) or *managed policies* (standalone policies attached to multiple identities). To learn how to choose between managed and inline policies, see Choose between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples include IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. You must specify a principal in a resource-based policy.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Other policy types

AWS supports additional policy types that can set the maximum permissions granted by more common policy types:

- **Permissions boundaries** – Set the maximum permissions that an identity-based policy can grant to an IAM entity. For more information, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs)** – Specify the maximum permissions for an organization or organizational unit in AWS Organizations. For more information, see Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – Set the maximum available permissions for resources in your accounts. For more information, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.
- **Session policies** – Advanced policies passed as a parameter when creating a temporary session for a role or federated user. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

AWS managed policies for Managed Integrations

To add permissions to users, groups, and roles, it is easier to use AWS managed policies than to write policies yourself. It takes time and expertise to create IAM customer managed policies that provide your team with only the permissions they need. To get started quickly, you can use our AWS managed policies. These policies cover common use cases and are available in your AWS account. For more information about AWS managed policies, see AWS managed policies in the *IAM User Guide*.

AWS services maintain and update AWS managed policies. You can't change the permissions in AWS managed policies. Services occasionally add additional permissions to an AWS managed policy to support new features. This type of update affects all identities (users, groups, and roles) where the policy is attached. Services are most likely to update an AWS managed policy when

a new feature is launched or when new operations become available. Services do not remove permissions from an AWS managed policy, so policy updates won't break your existing permissions.

Additionally, AWS supports managed policies for job functions that span multiple services. For example, the **ReadOnlyAccess** AWS managed policy provides read-only access to all AWS services and resources. When a service launches a new feature, AWS adds read-only permissions for new operations and resources. For a list and descriptions of job function policies, see AWS managed policies for job functions in the *IAM User Guide*.

AWS managed policy: AWSIoTManagedIntegrationsFullAccess

You can attach the `AWSIoTManagedIntegrationsFullAccess` policy to your IAM identities.

This policy grants full access permissions to Managed Integrations and related services. To view this policy in the AWS Management Console, see AWSIoTManagedIntegrationsFullAccess.

Permissions details

This policy includes the following permissions:

- `iotmanagedintegrations` – Provides full access to Managed Integrations and related services for the IAM users, groups, and roles you add this policy to.
- `iam` – Allows the assigned IAM users, groups, and roles to create a service-linked role in an AWS account.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iotmanagedintegrations:*",
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": "iam:CreateServiceLinkedRole",
```

```
      "Resource": "arn:aws:iam::*:role/aws-service-role/
iotmanagedintegrations.amazonaws.com/AWSServiceRoleForIoTManagedIntegrations",
      "Condition": {
        "StringEquals": {
          "iam:AWSServiceName": "iotmanagedintegrations.amazonaws.com"
        }
      }
    }
  ]
}
```

AWS managed policy: AWSIoTManagedIntegrationsRolePolicy

You can attach the `AWSIoTManagedIntegrationsRolePolicy` policy to your IAM identities.

This policy grants Managed Integrations permission to publish Amazon CloudWatch logs and metrics on your behalf.

To view this policy in the AWS Management Console, see AWSIoTManagedIntegrationsRolePolicy.

Permissions details

This policy includes the following permissions.

- `logs` – Provides ability to create Amazon CloudWatch log groups and stream logs to the groups.
- `cloudwatch` – Provides ability to publish Amazon CloudWatch metrics. For more information on Amazon CloudWatch metrics, see Metrics in Amazon CloudWatch.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "CloudWatchLogs",
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup"
      ],
      "Resource": [
        "arn:aws:logs:*:*:log-group:/aws/iotmanagedintegrations/*"
```

```
      ],
      "Condition": {
        "StringEquals": {
          "aws:PrincipalAccount": "${aws:ResourceAccount}"
        }
      }
    },
    {
      "Sid": "CloudWatchStreams",
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": [
        "arn:aws:logs:*:*:log-group:/aws/iotmanagedintegrations/*:log-stream:*"
      ],
      "Condition": {
        "StringEquals": {
          "aws:PrincipalAccount": "${aws:ResourceAccount}"
        }
      }
    },
    {
      "Sid": "CloudWatchMetrics",
      "Effect": "Allow",
      "Action": [
        "cloudwatch:PutMetricData"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "cloudwatch:namespace": [
            "AWS/IoTManagedIntegrations",
            "AWS/Usage"
          ]
        }
      }
    }
  ]
}
```

Managed Integrations updates to AWS managed policies

View details about updates to AWS managed policies for Managed Integrations since this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the Managed Integrations Document history page.

Change	Description	Date
Managed Integrations started tracking changes	Managed Integrations started tracking changes for its AWS managed policies.	March 03, 2025

How Managed Integrations works with IAM

Before you use IAM to manage access to Managed Integrations, learn what IAM features are available to use with Managed Integrations.

IAM features you can use with Managed Integrations

IAM feature	Managed Integrations support
Identity-based policies	Yes
Resource-based policies	No
Policy actions	Yes
Policy resources	Yes
Policy condition keys	Yes
ACLs	No
ABAC (tags in policies)	No
Temporary credentials	Yes
Principal permissions	Yes

IAM feature	Managed Integrations support
Service roles	Yes
Service-linked roles	Yes

To get a high-level view of how Managed Integrations and other AWS services work with most IAM features, see AWS services that work with IAM in the *IAM User Guide*.

Identity-based policies for Managed Integrations

Supports identity-based policies: Yes

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. To learn about all of the elements that you can use in a JSON policy, see IAM JSON policy elements reference in the *IAM User Guide*.

Identity-based policy examples for Managed Integrations

To view examples of Managed Integrations identity-based policies, see Identity-based policy examples for Managed Integrations.

Resource-based policies within Managed Integrations

Supports resource-based policies: No

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Policy actions for Managed Integrations

Supports policy actions: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Include actions in a policy to grant permissions to perform the associated operation.

To see a list of Managed Integrations actions, see Actions Defined by Managed Integrations in the *Service Authorization Reference*.

Policy actions in Managed Integrations use the following prefix before the action:

```
iot-mi
```

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [
      "iot-mi:action1",
      "iot-mi:action2"
         ]
```

To view examples of Managed Integrations identity-based policies, see Identity-based policy examples for Managed Integrations.

Policy resources for Managed Integrations

Supports policy resources: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. As a best practice, specify a resource using its Amazon Resource Name (ARN). For actions that don't support resource-level permissions, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

To see a list of Managed Integrations resource types and their ARNs, see Resources Defined by Managed Integrations in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see Actions Defined by Managed Integrations .

To view examples of Managed Integrations identity-based policies, see Identity-based policy examples for Managed Integrations.

Policy condition keys for Managed Integrations

Supports service-specific policy condition keys: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element specifies when statements execute based on defined criteria. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

To see a list of Managed Integrations condition keys, see Condition Keys for Managed Integrations in the *Service Authorization Reference*. To learn with which actions and resources you can use a condition key, see Actions Defined by Managed Integrations .

To view examples of Managed Integrations identity-based policies, see Identity-based policy examples for Managed Integrations.

ACLs in Managed Integrations

Supports ACLs: No

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

ABAC with Managed Integrations

Supports ABAC (tags in policies): Partial

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes called tags. You can attach tags to IAM entities and AWS resources, then design ABAC policies to allow operations when the principal's tag matches the tag on the resource.

To control access based on tags, you provide tag information in the condition element of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys.

If a service supports all three condition keys for every resource type, then the value is **Yes** for the service. If a service supports all three condition keys for only some resource types, then the value is **Partial**.

For more information about ABAC, see Define permissions with ABAC authorization in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see Use attribute-based access control (ABAC) in the *IAM User Guide*.

Using temporary credentials with Managed Integrations

Supports temporary credentials: Yes

Temporary credentials provide short-term access to AWS resources and are automatically created when you use federation or switch roles. AWS recommends that you dynamically generate temporary credentials instead of using long-term access keys. For more information, see Temporary security credentials in IAM and AWS services that work with IAM in the *IAM User Guide*.

Cross-service principal permissions for Managed Integrations

Supports forward access sessions (FAS): Yes

Forward access sessions (FAS) use the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. For policy details when making FAS requests, see Forward access sessions.

Service roles for Managed Integrations

Supports service roles: Yes

A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.

> **Warning**
>
> Changing the permissions for a service role might break Managed Integrations functionality. Edit service roles only when Managed Integrations provides guidance to do so.

Service-linked roles for Managed Integrations

Supports service-linked roles: Yes

A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

For details about creating or managing service-linked roles, see AWS services that work with IAM. Find a service in the table that includes a `Yes` in the **Service-linked role** column. Choose the **Yes** link to view the service-linked role documentation for that service.

Identity-based policy examples for Managed Integrations

By default, users and roles don't have permission to create or modify Managed Integrations resources. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see Create IAM policies (console) in the *IAM User Guide*.

For details about actions and resource types defined by Managed Integrations, including the format of the ARNs for each of the resource types, see Actions, Resources, and Condition Keys for Managed Integrations in the *Service Authorization Reference*.

Topics

- Policy best practices

- Using the Managed Integrations console
- Allow users to view their own permissions

Policy best practices

Identity-based policies determine whether someone can create, access, or delete Managed Integrations resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see Validate policies with IAM Access Analyzer in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Secure API access with MFA in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Using the Managed Integrations console

To access the Managed Integrations console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the Managed Integrations resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (users or roles) with that policy.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that they're trying to perform.

To ensure that users and roles can still use the Managed Integrations console, also attach the Managed Integrations *ConsoleAccess* or *ReadOnly* AWS managed policy to the entities. For more information, see Adding permissions to a user in the *IAM User Guide*.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
```

```
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Troubleshooting Managed Integrations identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Managed Integrations and IAM.

Topics

- I am not authorized to perform an action in Managed Integrations
- I am not authorized to perform iam:PassRole
- I want to allow people outside of my AWS account to access my Managed Integrations resources

I am not authorized to perform an action in Managed Integrations

If you receive an error that you're not authorized to perform an action, your policies must be updated to allow you to perform the action.

The following example error occurs when the `mateojackson` IAM user tries to use the console to view details about a fictional *`my-example-widget`* resource but doesn't have the fictional `iot-mi:`*`GetWidget`* permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform: iot-mi:GetWidget on resource: my-example-widget
```

In this case, the policy for the `mateojackson` user must be updated to allow access to the *`my-example-widget`* resource by using the `iot-mi:`*`GetWidget`* action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to Managed Integrations.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in Managed Integrations. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my Managed Integrations resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether Managed Integrations supports these features, see How Managed Integrations works with IAM.

- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.

Using service-linked roles for managed integrations

Managed integrations for AWS IoT Device Management uses AWS Identity and Access Management (IAM) service-linked roles. A service-linked role is a unique type of IAM role that is linked directly to managed integrations. Service-linked roles are predefined by managed integrations and include all the permissions that the service requires to call other AWS services on your behalf.

A service-linked role makes setting up managed integrations easier because you don’t have to manually add the necessary permissions. Managed integrations for AWS IoT Device Management defines the permissions of its service-linked roles, and unless defined otherwise, only managed integrations can assume its roles. The defined permissions include the trust policy and the permissions policy, and that permissions policy cannot be attached to any other IAM entity.

You can delete a service-linked role only after first deleting their related resources. This protects your managed integrations resources because you can't inadvertently remove permission to access the resources.

For information about other services that support service-linked roles, see AWS services that work with IAM and look for the services that have **Yes** in the **Service-linked roles** column. Choose a **Yes** with a link to view the service-linked role documentation for that service.

Service-linked role permissions for managed integrations

Managed integrations for AWS IoT Device Management uses the service-linked role named **AWSServiceRoleForIoTManagedIntegrations** – Provides managed integrations for AWS IoT Device Management permission to publish logs and metrics on your behalf.

The AWSServiceRoleForIoTManagedIntegrations service-linked role trusts the following services to assume the role:

- `iotmanagedintegrations.amazonaws.com`

The role permissions policy named AWSIoTManagedIntegrationsServiceRolePolicy allows managed integrations to complete the following actions on the specified resources:

- Action: `logs:CreateLogGroup, logs:DescribeLogGroups, logs:CreateLogStream, logs:PutLogEvents, logs:DescribeLogStreams, cloudwatch:PutMetricData` on `all of your managed integrations resources.`

JSON

```
{
  "Version":"2012-10-17",
  "Statement" : [
    {
      "Sid" : "CloudWatchLogs",
      "Effect" : "Allow",
      "Action" : [
        "logs:CreateLogGroup",
        "logs:DescribeLogGroups"
      ],
      "Resource" : [
        "arn:aws:logs:*:*:log-group:/aws/iotmanagedintegrations/*"
      ]
    },
    {
      "Sid" : "CloudWatchStreams",
      "Effect" : "Allow",
      "Action" : [
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams"
      ],
      "Resource" : [
        "arn:aws:logs:*:*:log-group:/aws/iotmanagedintegrations/*:log-stream:*"
      ]
    },
    {
      "Sid" : "CloudWatchMetrics",
      "Effect" : "Allow",
      "Action" : [
```

```
        "cloudwatch:PutMetricData"
      ],
      "Resource" : "*",
      "Condition" : {
        "StringEquals" : {
          "cloudwatch:namespace" : [
            "AWS/IoTManagedIntegrations",
            "AWS/Usage"
          ]
        }
      }
    }
  ]
}
```

You must configure permissions to allow your users, groups, or roles to create, edit, or delete a service-linked role. For more information, see Service-linked role permissions in the *IAM User Guide*.

Creating a service-linked role for managed integrations

You don't need to manually create a service-linked role. When you cause an event type such as calling the `PutRuntimeLogConfiguration`, `CreateEventLogConfiguration`, or `RegisterCustomEndpoint` API commands in the AWS Management Console, the AWS CLI, or the AWS API, managed integrations creates the service-linked role for you. For more information on `PutRuntimeLogConfiguration`, `CreateEventLogConfiguration`, or `RegisterCustomEndpoint`, see PutRuntimeLogConfiguration, CreateEventLogConfiguration, or RegisterCustomEndpoint.

If you delete this service-linked role, and then need to create it again, you can use the same process to recreate the role in your account. When you cause an event type such as calling the `PutRuntimeLogConfiguration`, `CreateEventLogConfiguration`, or `RegisterCustomEndpoint` API commands, managed integrations creates the service-linked role for you again. Alternatively, you can contact AWS Customer Support via the AWS Support Center Console. For more information on AWS Support Plans, see Compare AWS Support Plans.

You can also use the IAM console to create a service-linked role with the **IoT ManagedIntegrations - Managed Role** use case. In the AWS CLI or the AWS API, create a service-linked role with the `iotmanagedintegrations.amazonaws.com` service name. For more information, see Creating a service-linked role in the *IAM User Guide*. If you delete this service-linked role, you can use this same process to create the role again.

Editing a service-linked role for managed integrations

managed integrations does not allow you to edit the AWSServiceRoleForIoTManagedIntegrations service-linked role. After you create a service-linked role, you cannot change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM. For more information, see Editing a service-linked role in the *IAM User Guide*.

Deleting a service-linked role for managed integrations

If you no longer need to use a feature or service that requires a service-linked role, we recommend that you delete that role. That way you don't have an unused entity that is not actively monitored or maintained. However, you must clean up the resources for your service-linked role before you can manually delete it.

> **Note**
>
> If managed integrations is using the role when you try to delete the resources, then the deletion might fail. If that happens, wait for a few minutes and try the operation again.

To manually delete the service-linked role using IAM

Use the IAM console, the AWS CLI, or the AWS API to delete the AWSServiceRoleForIoTManagedIntegrations service-linked role. For more information, see Deleting a service-linked role in the *IAM User Guide*.

Supported Regions for managed integrations service-linked roles

Managed integrations for AWS IoT Device Management supports using service-linked roles in all of the Regions where the service is available. For more information, see AWS Regions and endpoints.

Use AWS Secrets Manager for data protection for C2C workflows

AWS Secrets Manager is a secret storage service that you can use to protect database credentials, API keys, and other secret information. Then in your code, you can replace hardcoded credentials with an API call to Secrets Manager. This helps ensure that the secret can't be compromised by

someone examining your code, because the secret isn't there. For an overview, see the AWS Secrets Manager User Guide.

Secrets Manager encrypts secrets using AWS Key Management Service keys. For more information, see Secret encryption and decryption in AWS Key Management Service.

Managed Integrations for AWS IoT Device Management integrates with AWS Secrets Manager so that you can store your data in Secrets Manager and use the secret ID in your configurations.

How managed integrations uses secrets

Open Authorization (OAuth) is an open standard for delegated access authorization, enabling users to grant websites or applications access to their information on other websites without sharing their passwords. It's a secure way for third-party applications to access user data on behalf of the user, providing a more secure alternative to sharing passwords.

In OAuth, a client ID and client secret are credentials that identify and authenticate a client application when it requests an access token.

Managed Integrations for AWS IoT Device Management uses OAuth to communicate with customers that use the C2C workflows. Customers need to provide the client ID and client secret to communicate. Managed Integrations customers will store a client ID and client secret in their AWS accounts, and managed integrations reads the client ID and client secret in our customer account.

How to create a secret

To create a secret, follow the steps in Create an AWS Secrets Manager secret in the AWS Secrets Manager User Guide.

You must create your secret with a customer-managed AWS KMS key for managed integrations to read the secret value. For more information, see Permissions for the AWS KMS key in the AWS Secrets Manager User Guide.

You must also use the IAM policies in the following section.

Grant access for managed integrations for AWS IoT Device Management to retrieve the secret

To allow managed integrations to retrieve the secret value from Secrets Manager, include the following permissions in the resource policy for the secret when you create it.

JSON

```
{
  "Version":"2012-10-17",
  "Statement" : [ {
    "Effect" : "Allow",
    "Principal" : {
      "Service" : "iotmanagedintegrations.amazonaws.com"
    },
    "Action" : [ "secretsmanager:GetSecretValue" ],
    "Resource" : "*" ,
    "Condition": {
        "StringEquals": {
          "aws:SourceArn": "arn:aws:iotmanagedintegrations:AWS
 Region:123456789012:account-association:account-association-id"

        }
      }
  } ]
}
```

Add the following statement to the policy for your customer-managed AWS KMS key.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [{
            "Effect": "Allow",
            "Action": [
                "kms:Decrypt",
                "kms:DescribeKey"
            ],
            "Principal": {
                "Service": [
                    "iotmanagedintegrations.amazonaws.com"
                ]
        },
            "Resource": [
            "arn:aws:kms:us-east-1:123456789012:key/*"
            ]
```

```
        }

    ]
}
```

Compliance validation for Managed Integrations

Our new AWS sign-up experience is not designed for regulated workloads. If you're using our new AWS sign-up experience, but you want to use AWS for regulated workloads, you can sign up for AWS (advanced) or activate advanced features for your AWS environment.

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. For more information about your compliance responsibility when using AWS services, see AWS Security Documentation.

Use Managed Integrations with interface VPC endpoints

You can establish a private connection between your Amazon VPC and AWS IoT Managed Integrations by creating an interface Amazon VPC endpoint. Interface endpoints are powered by AWS PrivateLink, a technology that enables you to privately access services by using private IP addresses. AWS PrivateLink restricts all network traffic between your VPC and IoT Managed Integrations to the Amazon network. You don't need an internet gateway, NAT device, or VPN connection.

You are not required to use AWS PrivateLink, but it's recommended. For more information about AWS PrivateLink and VPC endpoints, see Accessing AWS services through AWS PrivateLink in the *AWS PrivateLink Guide*.

Topics

- Considerations for AWS IoT Managed Integrations VPC endpoints

- Creating an interface VPC endpoint for AWS IoT Managed Integrations
- Testing your VPC endpoint
- Controlling access to services over VPC endpoints
- Pricing
- Limitations

Considerations for AWS IoT Managed Integrations VPC endpoints

Before you set up an interface VPC endpoint for AWS IoT Managed Integrations, review Interface endpoint properties and limitations in the *AWS PrivateLink Guide*.

AWS IoT Managed Integrations supports making calls to all of its API actions from your VPC through interface VPC endpoints.

Supported endpoints

AWS IoT Managed Integrations supports VPC endpoints for the following service interfaces:

- **Control plane API**: `com.amazonaws.region.iotmanagedintegrations.api`

Unsupported endpoints

The following AWS IoT Managed Integrations endpoints do not support VPC endpoints:

- **MQTT endpoints**: MQTT devices are typically deployed in end-user environments rather than within AWS VPCs, making AWS PrivateLink integration unnecessary.
- **OAuth callback endpoints**: Many third-party platforms do not operate within AWS infrastructure, reducing the benefits of AWS PrivateLink support for OAuth flows.

Availability

AWS IoT Managed Integrations VPC endpoints are available in the following AWS Regions:

- Canada (Central) - ca-central-1
- Europe (Ireland) - eu-west-1

Additional regions will be supported as AWS IoT Managed Integrations expands its availability.

Dual-stack support

AWS IoT Managed Integrations VPC endpoints support both IPv4 and IPv6 traffic. You can create VPC endpoints with the following IP address types:

- **IPv4**: Assigns IPv4 addresses to endpoint network interfaces
- **IPv6**: Assigns IPv6 addresses to endpoint network interfaces (requires IPv6-only subnets)
- **Dualstack**: Assigns both IPv4 and IPv6 addresses to endpoint network interfaces

Creating an interface VPC endpoint for AWS IoT Managed Integrations

You can create a VPC endpoint for the AWS IoT Managed Integrations service using either the Amazon VPC Console or the AWS CLI (AWS CLI).

To create an interface VPC endpoint for AWS IoT Managed Integrations (console)

1. Open the Amazon VPC Console at Amazon VPC Console.
2. In the navigation pane, choose **Endpoints**.
3. Choose **Create endpoint**.
4. For **Service category**, choose **AWS services**.
5. For **Service name**, select the service name that corresponds to your AWS Region. For example:
   - `com.amazonaws.ca-central-1.iotmanagedintegrations.api`
   - `com.amazonaws.eu-west-1.iotmanagedintegrations.api`
6. For **VPC**, select the VPC from which you'll access AWS IoT Managed Integrations.
7. For **Additional settings**, **Enable DNS name** is selected by default. We recommend that you keep this setting. This ensures that requests to the AWS IoT Managed Integrations public service endpoints resolve to your Amazon VPC endpoint.
8. For **Subnets**, select the subnets in which to create endpoint network interfaces. You can select one subnet per Availability Zone.
9. For **IP address type**, choose from the following options:
   - **IPv4**: Assign IPv4 addresses to the endpoint network interfaces
   - **IPv6**: Assign IPv6 addresses to the endpoint network interfaces (supported only if all selected subnets are IPv6-only)
   - **Dualstack**: Assign both IPv4 and IPv6 addresses to the endpoint network interfaces

10. For **Security groups**, select the security groups to associate with the endpoint network interfaces. The security group rules must allow communication between the endpoint network interface and the resources in your VPC that communicate with the service.
11. For **Policy**, choose **Full access** to allow all operations by all principals on all resources over the interface endpoint. To restrict access, choose **Custom** and specify a policy.
12. (Optional) To add a tag, choose **Add new tag** and enter the tag key and value.
13. Choose **Create endpoint**.

To create an interface VPC endpoint for IoT Managed Integrations (AWS CLI)

Use the create-vpc-endpoint command and specify the VPC ID, VPC endpoint type (interface), service name, subnets that will use the endpoint, and security groups to associate with the endpoint network interfaces.

```
aws ec2 create-vpc-endpoint \
  --vpc-id vpc-12345678 \
  --route-table-ids rtb-12345678 \
  --service-name com.amazonaws.ca-central-1.iotmanagedintegrations.api \
  --vpc-endpoint-type Interface \
  --subnet-ids subnet-12345678 subnet-87654321 \
  --security-group-ids sg-12345678
```

Testing your VPC endpoint

After you create your VPC endpoint, you can test the connection by making API calls to AWS IoT Managed Integrations from an EC2 instance in your VPC.

Prerequisites

- An EC2 instance in a private subnet within your VPC
- Appropriate IAM permissions for AWS IoT Managed Integrations operations
- Security group rules that allow HTTPS traffic (port 443) to the VPC endpoint

Testing the connection

1. Connect to your Amazon EC2 instance in the private subnet.
2. Verify DNS resolution for the private DNS name:

```
dig api.iotmanagedintegrations.region.api.aws
```

3. Test HTTPS connectivity:

```
curl -v https://api.iotmanagedintegrations.region.api.aws
```

4. Make an AWS IoT Managed Integrations API call:

```
aws iot-managed-integrations list-destinations \
  --region region \
  --endpoint-url https://api.iotmanagedintegrations.region.api.aws
```

Replace `region` with your AWS Region (for example, `ca-central-1`).

Controlling access to services over VPC endpoints

A VPC endpoint policy is an IAM resource policy that you attach to an interface VPC endpoint when you create or modify the endpoint. If you don't attach a policy when you create an endpoint, we attach a default policy for you that allows full access to the service. An endpoint policy doesn't override or replace IAM user policies or service-specific policies. It's a separate policy for controlling access from the endpoint to the specified service.

Endpoint policies must be written in JSON format. For more information, see Controlling access to services with VPC endpoints in the *Amazon VPC User Guide*.

Example: VPC endpoint policy for AWS IoT Managed Integrations actions

The following is an example of a VPC endpoint policy for AWS IoT Managed Integrations. This policy allows users connecting to AWS IoT Managed Integrations through the VPC endpoint to access destinations but denies access to credential lockers.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": [
                "iotmanagedintegrations:ListDestinations",
```

```
                "iotmanagedintegrations:GetDestination",
                "iotmanagedintegrations:CreateDestination",
                "iotmanagedintegrations:UpdateDestination",
                "iotmanagedintegrations:DeleteDestination"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action": [
                "iotmanagedintegrations:ListCredentialLockers",
                "iotmanagedintegrations:GetCredentialLocker",
                "iotmanagedintegrations:CreateCredentialLocker",
                "iotmanagedintegrations:DeleteCredentialLocker"
            ],
            "Resource": "*"
        }
    ]
}
```

Example: VPC endpoint policy that restricts access to a specific IAM role

The following example of a VPC endpoint policy allows access to AWS IoT Managed Integrations only for IAM principals that have the specified IAM role (`IoTManagedIntegrationsVPCRole`) in their trust chain. All other IAM principals are denied access.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "*",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:PrincipalArn": "arn:aws:iam::123456789012:role/
IoTManagedIntegrationsVPCRole"
                }
            }
        }
    ]
```

```
}
```

Pricing

You are charged standard rates for creating and using an interface VPC endpoint with AWS IoT Managed Integrations. For more information, see AWS PrivateLink pricing.

Limitations

- The CreateAccountAssociation API, is designed to perform OAuth with third-party cloud services, which requires the request to leave the Amazon network. This is important for customers using AWS PrivateLink to contain their traffic within the VPC, as AWS PrivateLink cannot provide complete end-to-end containment for this API call.
- VPC endpoints for AWS IoT Managed Integrations are not available in AWS GovCloud (US) Regions.

For general VPC endpoint limitations, see Interface endpoint properties and limitations in the *Amazon VPC User Guide*.

Connect to managed integrations for AWS IoT Device Management FIPS endpoints

AWS IoT provides a control plane endpoint that support the Federal Information Processing Standard (FIPS) 140-2. FIPS compliant endpoints are different from standard AWS endpoints. To interact with managed integrations for AWS IoT Device Management in a FIPS-compliant manner, you must use the endpoints described below with your FIPS compliant client. The AWS IoT console is not FIPS compliant.

The following sections describe how to access the FIPS compliant AWS IoT endpoint by using the REST API, an SDK, or the AWS CLI.

Control plane endpoints

The FIPS compliant **control plane** endpoints that support the managed integrations operations and their related AWS CLI commands are listed in FIPS Endpoints by Service. In FIPS Endpoints by Service, find the **AWS IoT Device Management - Managed integrations** service, and look up the endpoint for your AWS Region.

To use the FIPS compliant endpoint when you access themanaged integrations operations, use the AWS SDK or the REST API with the endpoint that is appropriate for your AWS Region.

To use the FIPS compliant endpoint when you run managed integrations CLI commands, add the **--endpoint** parameter with the appropriate endpoint for your AWS Region to the command.

Configuring logging for Managed Integrations

Managed Integrations provides two types of logs delivered to Amazon CloudWatch Logs in your account:

- *Event logs* – Cloud-side logs capturing events from Managed Integrations workflows. Written to the `/aws/iotmanagedintegrations/EventLog` log group.
- *Runtime logs* – Device-side logs published by your devices or hubs. Written to the `/aws/iotmanagedintegrations/RuntimeLog` log group.

Supported resource types

Each resource type corresponds to specific Managed Integrations workflows:

Resource type	Workflows
`managed-thing`	Provisioning, commands, event processing
`credential-locker`	Credential locker operations
`provisioning-profile`	Provisioning profile operations
`ota-task`	OTA update tasks
`account-association`	Third-party discovery failures, OAuth callback failures, token refresh failures

Creating event log configurations

Use the CreateEventLogConfiguration API action to enable event logging for a resource type. We recommend creating a separate configuration for each resource type, using `*` as the resource identifier to capture logs for all resources of that type. Start at the ERROR log level to receive all failure logs without generating excessive volume or cost.

```
aws iot-managed-integrations create-event-log-configuration \
    --resource-type "managed-thing" \
```

```
    --resource-id "*" \
    --event-log-level "ERROR"
```

Repeat for each resource type:

```
aws iot-managed-integrations create-event-log-configuration \
    --resource-type "credential-locker" \
    --resource-id "*" \
    --event-log-level "ERROR"

aws iot-managed-integrations create-event-log-configuration \
    --resource-type "provisioning-profile" \
    --resource-id "*" \
    --event-log-level "ERROR"

aws iot-managed-integrations create-event-log-configuration \
    --resource-type "ota-task" \
    --resource-id "*" \
    --event-log-level "ERROR"

aws iot-managed-integrations create-event-log-configuration \
    --resource-type "account-association" \
    --resource-id "*" \
    --event-log-level "ERROR"
```

After you call `CreateEventLogConfiguration`, logs are pushed to the `/aws/iotmanagedintegrations/EventLog` log group in CloudWatch Logs. Use ListEventLogConfigurations to view all configurations, or GetEventLogConfiguration to retrieve a specific configuration by ID.

Updating event log configurations

You can change the log level at any time using UpdateEventLogConfiguration:

```
aws iot-managed-integrations update-event-log-configuration \
    --id "your-configuration-id" \
    --event-log-level "DEBUG"
```

To remove an event log configuration, use DeleteEventLogConfiguration.

> **Important**
>
> Enabling DEBUG logging generates significantly more log entries and increases CloudWatch Logs costs. We recommend starting with ERROR and only increasing the level when actively troubleshooting an issue.

Configuring runtime logs

Runtime log configurations control device-side logging behavior for individual managed things. Use PutRuntimeLogConfiguration to set the runtime log configuration for a specific managed thing:

```
aws iot-managed-integrations put-runtime-log-configuration \
    --managed-thing-id "your-managed-thing-id" \
    --runtime-log-configurations
 '{"LogLevel":"ERROR","UploadLog":true,"UploadPeriodMinutes":5}'
```

Use GetRuntimeLogConfiguration to retrieve the current configuration, or ResetRuntimeLogConfiguration to reset it to defaults.

Setting up alarms on error logs

We recommend setting up a monitoring system on your ERROR logs to alert you of consistent failures. You can use Amazon CloudWatch metric filters and alarms to detect error patterns automatically.

To create an alarm on error logs

1. Create a metric filter on the `/aws/iotmanagedintegrations/EventLog` log group that matches error-level log entries.
2. Create a CloudWatch alarm on the metric filter that triggers when the error count exceeds your threshold.
3. Configure an Amazon Simple Notification Service topic as the alarm action to receive notifications.

For detailed instructions, see Using Amazon CloudWatch alarms in the *Amazon CloudWatch User Guide*.

Monitoring Managed Integrations

Monitoring is an important part of maintaining the reliability, availability, and performance of Managed Integrations and your other AWS solutions. AWS provides the following monitoring tools to watch Managed Integrations, report when something is wrong, and take automatic actions when appropriate:

- *AWS CloudTrail* captures API calls and related events made by or on behalf of your AWS account and delivers the log files to an Amazon S3 bucket that you specify. You can identify which users and accounts called AWS, the source IP address from which the calls were made, and when the calls occurred. For more information, see the AWS CloudTrail User Guide.

Logging Managed Integrations API calls using AWS CloudTrail

Managed Integrations is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service. CloudTrail captures all API calls for Managed Integrations as events. The calls captured include calls from the Managed Integrations console and code calls to the Managed Integrations API operations. Using the information collected by CloudTrail, you can determine the request that was made to Managed Integrations, the IP address from which the request was made, when it was made, and additional details.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root user or user credentials.
- Whether the request was made on behalf of an IAM Identity Center user.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

CloudTrail is active in your AWS account when you create the account and you automatically have access to the CloudTrail **Event history**. The CloudTrail **Event history** provides a viewable, searchable, downloadable, and immutable record of the past 90 days of recorded management events in an AWS Region. For more information, see Working with CloudTrail Event history in the *AWS CloudTrail User Guide*. There are no CloudTrail charges for viewing the **Event history**.

For an ongoing record of events in your AWS account past 90 days, create a trail or a CloudTrail Lake event data store.

CloudTrail trails

A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. All trails created using the AWS Management Console are multi-Region. You can create a single-Region or a multi-Region trail by using the AWS CLI. Creating a multi-Region trail is recommended because you capture activity in all AWS Regions in your account. If you create a single-Region trail, you can view only the events logged in the trail's AWS Region. For more information about trails, see Creating a trail for your AWS account and Creating a trail for an organization in the *AWS CloudTrail User Guide*.

You can deliver one copy of your ongoing management events to your Amazon S3 bucket at no charge from CloudTrail by creating a trail, however, there are Amazon S3 storage charges. For more information about CloudTrail pricing, see AWS CloudTrail Pricing. For information about Amazon S3 pricing, see Amazon S3 Pricing.

CloudTrail Lake event data stores

CloudTrail Lake lets you run SQL-based queries on your events. CloudTrail Lake converts existing events in row-based JSON format to Apache ORC format. ORC is a columnar storage format that is optimized for fast retrieval of data. Events are aggregated into *event data stores*, which are immutable collections of events based on criteria that you select by applying advanced event selectors. The selectors that you apply to an event data store control which events persist and are available for you to query. For more information about CloudTrail Lake, see Working with AWS CloudTrail Lake in the *AWS CloudTrail User Guide*.

CloudTrail Lake event data stores and queries incur costs. When you create an event data store, you choose the pricing option you want to use for the event data store. The pricing option determines the cost for ingesting and storing events, and the default and maximum retention period for the event data store. For more information about CloudTrail pricing, see AWS CloudTrail Pricing.

Management events in CloudTrail

Management events provide information about management operations that are performed on resources in your AWS account. These are also known as control plane operations. By default, CloudTrail logs management events.

Managed Integrations logs the following Managed Integrations control plane operations to CloudTrail as *management events*.

- `CreateCloudConnector`
- `UpdateCloudConnector`
- `GetCloudConnector`
- `DeleteCloudConnector`
- `ListCloudConnectors`
- `CreateConnectorDestination`
- `UpdateConnectorDestination`
- `GetConnectorDestination`
- `DeleteConnectorDestination`
- `ListConnectorDestinations`
- `CreateAccountAssociation`
- `UpdateAccountAssociation`
- `GetAccountAssociation`
- `DeleteAccountAssociation`
- `ListAccountAssociations`
- `StartAccountAssociationRefresh`
- `ListManagedThingAccountAssociations`
- `RegisterAccountAssociation`
- `DeregisterAccountAssociation`
- `SendConnectorEvent`
- `ListDeviceDiscoveries`
- `ListDiscoveredDevices`

Event examples

An event represents a single request from any source and includes information about the requested API operation, the date and time of the operation, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so events don't appear in any specific order.

The following example shows a CloudTrail event that demonstrates a successful `CreateCloudConnector` API operation.

Successful CloudTrail event with the `CreateCloudConnector` API operation.


```
{
    "eventVersion": "1.09",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "EXAMPLE",
        "arn": "arn:aws:sts::111122223333:assumed-role/Admin/EXAMPLE",
        "accountId": "111122223333",
        "accessKeyId": "EXAMPLEKYSBQSCGRIC",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROAZOZQFKYSFZVB2J2GN",
                "arn": "arn:aws:iam::111122223333:role/Admin",
                "accountId": "111122223333",
                "userName": "Admin"
            },
            "attributes": {
                "creationDate": "2025-06-05T18:26:16Z",
                "mfaAuthenticated": "false"
            }
        }
    },
    "eventTime": "2025-06-05T18:30:40Z",
    "eventSource": "iotmanagedintegrations.amazonaws.com",
    "eventName": "CreateCloudConnector",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "192.0.2.0",
    "userAgent": "PostmanRuntime/7.44.0",
    "requestParameters": {
        "EndpointType": "LAMBDA",
        "Description": "Manual testing for C2C CT Validation",
        "ClientToken": "abc7460",
        "EndpointConfig": {
            "lambda": {
                "arn": "arn:aws:lambda:us-
east-1:111122223333:function:LightweightMockConnector7460"
            }
        },
        "Name": "EdenManualTestCloudConnector"
```

```
    },
    "responseElements": {
        "X-Frame-Options": "DENY",
        "Access-Control-Expose-Headers": "Content-Length,Content-Type,X-Amzn-
Errortype,X-Amzn-Requestid",
        "Strict-Transport-Security": "max-age:47304000; includeSubDomains",
        "Cache-Control": "no-store, no-cache",
        "X-Content-Type-Options": "nosniff",
        "Content-Security-Policy": "upgrade-insecure-requests; default-src 'none';
 object-src 'none'; frame-ancestors 'none'; base-uri 'none'",
        "Pragma": "no-cache",
        "Id": "f7e633e719404c4a933596b4d0cc276e",
        "Arn": "arn:aws:iotmanagedintegrations:us-east-1:111122223333:cloud-connector/
EXAMPLE404c4a933596b4d0cc276e"
    },
    "requestID": "c0071fd1-b8e0-400a-bcc0-EXAMPLE9e4",
    "eventID": "95b318ea-2f63-4183-9c22-EXAMPLE3e",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "111122223333",
    "eventCategory": "Management"
}
```

The following example shows a CloudTrail event that demonstrates a successful `ListDiscoveredDevices` API operation.

Successful CloudTrail event with the `ListDiscoveredDevices` API operation.

```
{
    "eventVersion": "1.09",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "EZAMPLE",
        "arn": "arn:aws:sts::444455556666:assumed-role/Admin/EXAMPLE",
        "accountId": "444455556666",
        "accessKeyId": "EXAMPLERJ26PYMH",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "EXAMPLE",
                "arn": "arn:aws:iam::444455556666:role/Admin",
                "accountId": "444455556666",
```

```
                "userName": "Admin"
            },
            "attributes": {
                "creationDate": "2025-06-10T23:37:31Z",
                "mfaAuthenticated": "false"
            }
        }
    },
    "eventTime": "2025-06-10T23:38:07Z",
    "eventSource": "iotmanagedintegrations.amazonaws.com",
    "eventName": "ListDiscoveredDevices",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "192.0.2.0",
    "userAgent": "EXAMPLE-runtime/2.4.0",
    "requestParameters": {
        "Identifier": "EXAMPLE4f268483a17d8060f014"
    },
    "responseElements": null,
    "requestID": "27ae1f61-e2e6-43e4-bf17-EXAMPLEa568",
    "eventID": "34734e81-76a8-49a4-9641-EXAMPLE28ed",
    "readOnly": true,
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "444455556666",
    "eventCategory": "Management"
}
```


For information about CloudTrail record contents, see CloudTrail record contents in the *AWS CloudTrail User Guide*.

Document history for the Managed Integrations Developer Guide

The following table describes the documentation releases for Managed Integrations.

Change	Description	Date
Updated notifications and device discovery documentation	Added the `iotMiArrivalTimestamp` field to the `DEVICE_EVENT` notification payload. Documented how `Brand` and `Model` fields are populated for C2C devices (via `DeviceMetadata` in the `DEVICE_DISCOVERY` event) and for hub-connected devices (from the Matter Basic Information cluster).	July 2, 2026
General availability release	General availability release of the Managed Integrations Developer Guide	June 25, 2025
Initial preview release	Initial preview release of the Managed Integrations Developer Guide	March 3, 2025