IoT Device Access

Developer Guide

Issue 1.0

Date 2025-07-29

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are the property of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei Cloud and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Contents

1 Before You Start

Overview

To create an IoT solution based on Huawei Cloud IoTDA, perform the operations described in the table below.

Operation	Description
Product Development	Manage products, develop product models and codecs, and perform online debugging on the IoT Device Access (IoTDA) console.
Development on the Application Side	Carry out development for connection between applications and the platform, including calling APIs, obtaining service data, and managing HTTPS certificates.
Development on the Device Side	Carry out development for connection between devices and the platform, including connecting devices to the platform, reporting service data to the platform, and processing commands delivered by the platform.

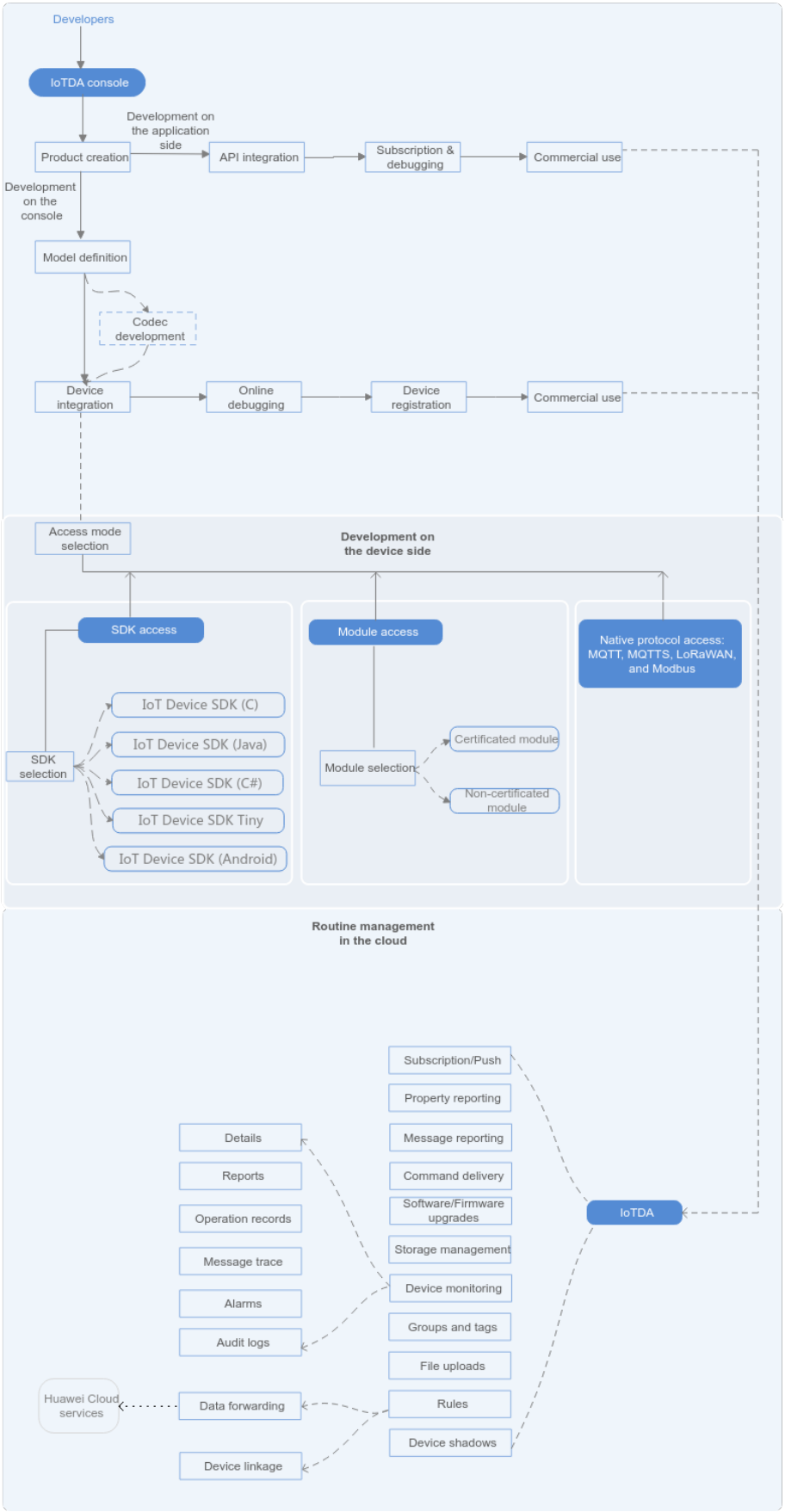
Service Process

The following describes the complete process of using IoTDA, including product development, device-side development, application-side development, and routine management.

- Product development: You can perform development operations on the IoTDA console. For example, you can create a product or device, develop a product model or codec, and perform online debugging.
- Application-side development: The platform provides robust device management capabilities through APIs. You can develop applications based on the APIs to meet requirements in different industries such as smart city, smart campus, smart industry, and IoV.
- Device-side development: You can connect devices to the platform by integrating SDKs or modules, or using native protocols.

- Routine management: After a physical device is connected, you can perform routine device management on the IoTDA console or by calling APIs.

Figure 1-1 Flowchart



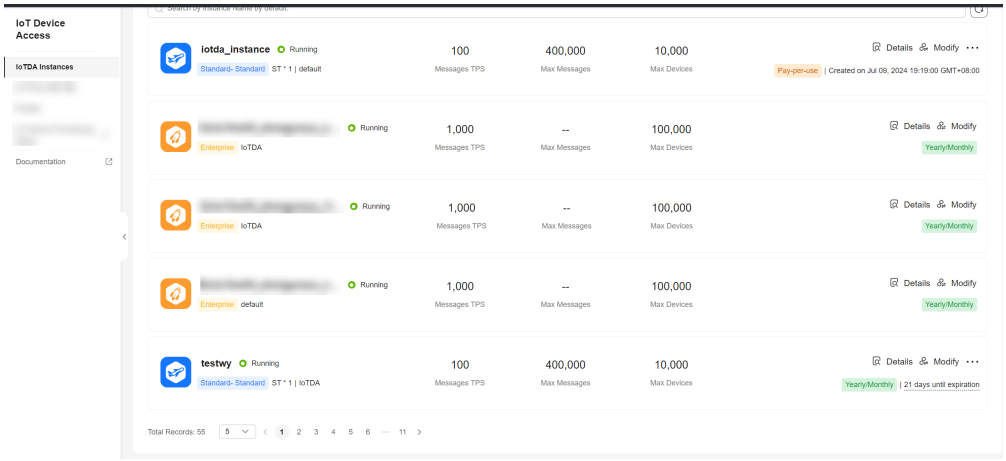
FAQ

2 Obtaining Resources

Platform Connection Information

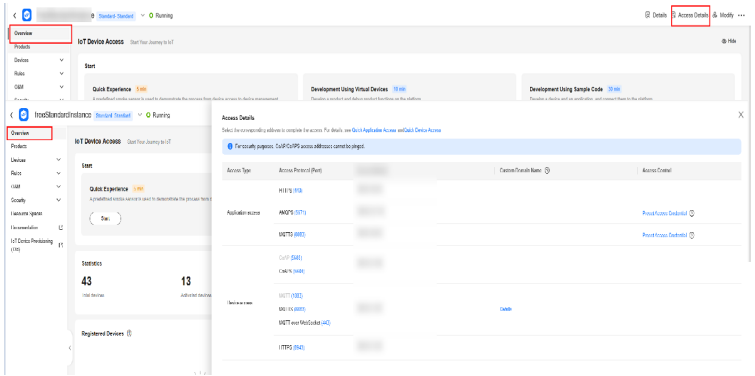
1. Log in to the **IoTDA console**. In the navigation pane, choose **IoTDA Instances**, and click the target instance card.

Figure 2-1 Instance management - Changing instance



2. In the navigation pane, choose **Overview**. In the **Instance Information** area, click **Access Details**.

Figure 2-2 Obtaining access information



Device Development Resources

You can connect devices to IoTDA using MQTT, LwM2M/CoAP, and HTTPS, as well as connect devices that use Modbus, OPC UA, and OPC DA through IoT Edge. You can also connect devices to IoTDA by calling APIs or integrating SDKs.

Resource Package	Description	Download Link
IoT Device Java SDK	Devices can connect to the platform by integrating the IoT Device Java SDK. The demo provides the sample code for calling SDK APIs. For details, see **IoT Device Java SDK**.	**IoT Device Java SDK**
IoT Device C SDK for Linux/Windows	Devices can connect to the platform by integrating the IoT Device C SDK. The demo provides the sample code for calling SDK APIs. For details, see **IoT Device C SDK**.	**IoT Device C SDK for Linux/Windows**
IoT Device C# SDK	Devices can connect to the platform by integrating the IoT Device C# SDK. The demo provides the sample code for calling SDK APIs. For details, see **IoT Device C# SDK**.	**IoT Device C# SDK**
IoT Device Android SDK	Devices can connect to the platform by integrating the IoT Device Android SDK. The demo provides the sample code for calling SDK APIs. For details, see **IoT Device Android SDK**.	**IoT Device Android SDK**
Device IoT Device Go SDK (Community Edition)	Devices can connect to the platform by integrating the IoT Device Go SDK. The demo provides the code sample for calling the SDK APIs. For details, see **IoT Device Go SDK**.	**IoT Device Go SDK (Community Edition)**

Resource Package	Description	Download Link
IoT Device Python SDK	Devices can connect to the platform by integrating the IoT Device Python SDK. The demo provides the code sample for calling the SDK APIs. For details, see **IoT Device Python SDK**.	**IoT Device Python SDK**
IoT Device Tiny C SDK for Linux/Windows	Devices can connect to the platform by integrating the IoT Device Tiny C SDK. The demo provides the sample code for calling SDK APIs. For details, see **IoT Device Tiny C SDK for Linux/Windows**.	**IoT Device Tiny C SDK for Linux/Windows**
IoT Device ArkTS (OpenHarmony) SDK	Devices can connect to the platform by integrating the IoT Device ArkTS SDK. The demo provides the code sample for calling the SDK APIs. For details, see **IoT Device ArkTS (OpenHarmony) SDK**.	**IoT Device ArkTS (OpenHarmony) SDK**
Native MQTT or MQTTS access	Devices can be connected to the platform using the native MQTT or MQTTS protocol. The demo provides the sample code for SSL-encrypted link setup, TCP link setup, data reporting, and topic subscription. Examples: **Java**, **Python**, **Android**, **C**, **C#**, and **Node.js**	**quickStart(Java)** **quickStart(Android)** **quickStart(Python)** **quickStart(C)** **quickStart(C#)** **quickStart(Node.js)**

Resource Package	Description	Download Link
Product model template	Product model templates of typical scenarios are provided. You can customize product models based on the templates. For details, see **Developing a Product Model Offline**.	**Product Model Example**
Codec example	Demo codec projects are provided for you to perform secondary development.	**Codec Example**
Codec test tool	The tool is used to check whether the codec developed offline is normal.	**Codec Test Tool**
NB-IoT device simulator	The tool is used to simulate the access of NB-IoT devices to the platform using LwM2M over CoAP for data reporting and command delivery. For details, see **Connecting and Debugging an NB-IoT Device Simulator**.	**NB-IoT Device Simulator**

Application Development Resources

The platform provides a wealth of application-side APIs to ease application development. Applications can call these APIs to implement services such as secure access, device management, data collection, and command delivery.

Resource Package	Description	Download Link
Application API Java Demo	You can call **application-side APIs** to experience service functions and service processes.	**API Java Demo**

Resource Package	Description	Download Link
Application Java SDK	You can use Java methods to call **application-side APIs** to communicate with the platform. For details, see **Java SDK**.	**Application Java SDK**
Application .NET SDK	You can use .NET methods to call **application-side APIs** to communicate with the platform. For details, see **.NET SDK**.	**Application .NET SDK**
Application Python SDK	You can use Python methods to call **application-side APIs** to communicate with the platform. For details, see **Python SDK**.	**Application Python SDK**
Application Go SDK	You can use Go methods to call **application-side APIs** to communicate with the platform. For details, see **Go SDK**.	**Application Go SDK**
Application Node.js SDK	You can use Node.js methods to call **application-side APIs** to communicate with the platform. For details, see **Node.js SDK**.	**Application Node.js SDK**
Application PHP SDK	You can use PHP methods to call **application-side APIs** to communicate with the platform. For details, see **PHP SDK**.	**Application PHP SDK**

Certificates

The following certificates are used when devices and applications need to verify IoTDA.

NOTE

- The certificates apply only to Huawei Cloud IoTDA and must be used together with the corresponding domain name.
- CA certificates can no longer be used to verify server certificates upon expiration. Replace CA certificates before they expire to ensure that devices can connect to the IoT platform properly.

Table 2-1 Certificates

Certificate Package Name	Region and Edition	Certificate Type	Certificate Format	Description	Download Link
certificate	CN-Hong Kong, AP-Singapore, AP-Bangkok, AP-Jakarta, AF-Johannesburg, LA-Santiago, LA-Sao Paulo1, LA-Mexico City2, and ME-Riyadh	Device certificate	pem, jks, and bks	Used by a device to verify the platform identity. The certificate must be used together with the device access domain name.	**Certificate file**

Certificate Package Name	Region and Edition	Certificate Type	Certificate Format	Description	Download Link
certificate	CN-Hong Kong, AP-Singapore, AP-Bangkok, AP-Jakarta, AF-Johannesburg, LA-Santiago, LA-Sao Paulo1, LA-Mexico City2, and ME-Riyadh	Application certificate	pem, jks, and bks	Application access: HTTPS/AMQPS/MQTTS platform CA certificates	**Certificate file**

3 Development on the Device Side

3.1 Device Access

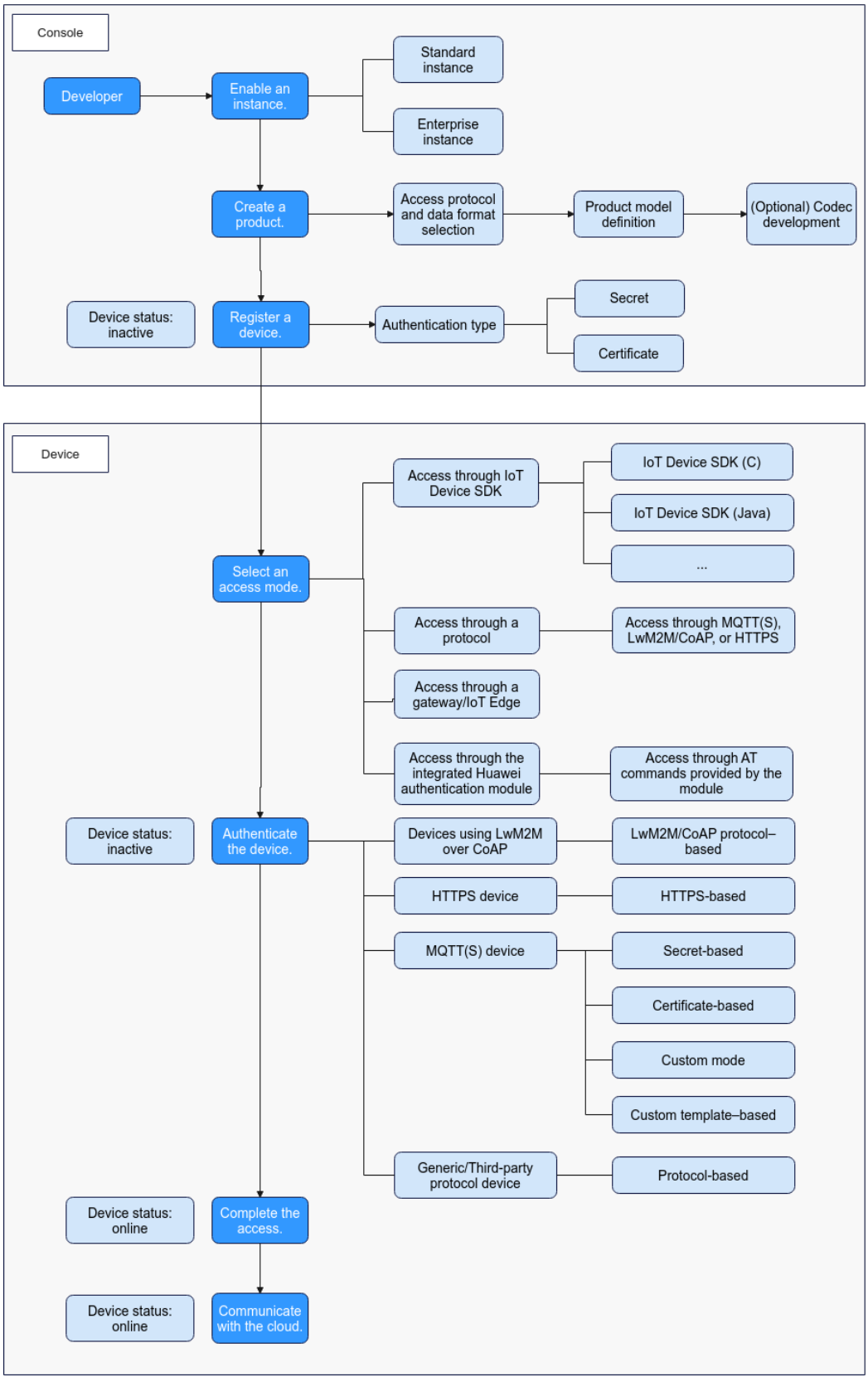
Process

You can use various protocols to access Huawei Cloud IoTDA, including:

- Common **native protocols**: MQTT(S), HTTPS, and LwM2M/CoAP(S)
- Standard protocols for access through gateways or IoT Edge: Modbus, OPC UA, OPC DA, ONVIF, GB28181, and LoRa
- Common protocols in some industries: JT808 (vehicle terminal communication protocol), SL651 (hydrological monitoring data communication protocol), and HJ212 (environmental protection industry data transmission standard protocol)
- TCP proprietary protocols and third-party protocols

For more protocols, see **Device Access Protocols**.

Figure 3-1 Device access development process



TLS

IoTDA supports Transport Layer Security (TLS) for encrypted communication and secure client connections. When TLS is utilized, clients can transmit the Server Name Indication (SNI) and access domain name during connection establishment with the device, which is essential for features like **custom domain names**, **device self-registration**, and **custom authentication**.

Table 3-1 TLS types supported by common protocols

Protocol	Operations Supported	Supported TLS Version	Port
MQTT	Publish/ Subscribe	Not applicable	1883
MQTTS	Publish/ Subscribe	1.1, 1.2, and 1.3	8883
MQTT over WebSocket (WSS)	Publish/ Subscribe	TLS 1.2	443
HTTPS	Publish	TLS 1.2	443
CoAP	Report and deliver	Not applicable	5683
CoAPS	Report and deliver	DTLS 1.2	5684

Access via Device-side SDKs

IoTDA offers device SDKs for seamless integration with Huawei Cloud, supporting functions like file uploading/downloading, automatic reconnection, OTA upgrades, data reporting, and time synchronization. The SDKs are available in C, C#, Java, Android, Go, Python, and ArkTS for HarmonyOS development. For details, see **Device SDKs**.

Access via Native Protocols

Devices can connect to IoTDA using native protocols such as MQTT(S), HTTPS, CoAP(S), or LwM2M. When a device employs the binary format, its data must be converted between binary and JSON formats using the **codec** deployed on the platform for communication with IoTDA.

Table 3-2 Native protocols

Protocol	Operations Supported	Transport Layer	Power Consumption	Applicable Network	Feature	Common Usage Scenario
MQTT(S)	Upstream and downstream	TCP	Low	Unstable/ High-latency	Lightweight and low power consumption; publish/subscribe model for one-to-many communication; persistent sessions	Recommended industry protocol for persistent connection scenarios. It can be used in IoT systems that require bidirectional communication, device control, or high scalability, such as smart city, Internet of Vehicles (IoV), energy, electric power, and Industry 4.0 solution.
HTTPS	Upstream only	TCP	High	Stable and high-bandwidth	Various data formats available; one-way communication for client-intiated requests; stateless with independent requests	Scenarios where data is integrated with existing web services (such as apps and web pages) or requires high readability.
CoAP(S)/ LwM2M	Upstream and downstream	UDP	Very low	Extremely low bandwidth /High packet loss rate	Designed for restricted devices; lightweight and multicast; low costs; binary format (CBOR)	This technology is commonly employed on low-power devices with limited resources, such as water meters and electricity meters, as well as on devices with extremely restricted resources like battery-powered sensors or those operating solely on UDP networks.

Access via Huawei-certified Modules

Huawei-certified modules are integrated with the **IoT Device SDK Tiny** and have passed Huawei certification tests. They comply with **Huawei AT command specifications**. You can send and receive data with a few clicks using AT commands, greatly reducing device interconnection workload and device commissioning period.

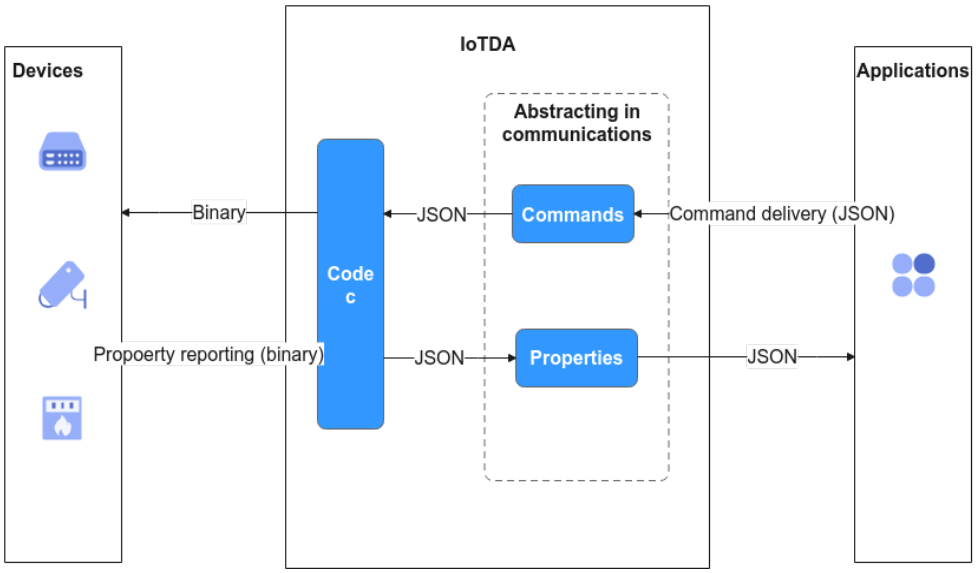
3.2 Product Development

3.2.1 Product Development Guide

In the IoT platform integration solution, the IoT platform provides open APIs for applications to connect devices that use various protocols. To better manage devices, the IoT platform needs to understand the device capabilities and the formats of data reported by devices. Therefore, you need to develop product models and codecs on the IoT platform.

- A **product model** is a JSON file that describes device capabilities. It defines basic device properties and message formats for data reporting and command delivery. To define a product model is to construct an abstract model of a device in the platform to enable the platform to understand the device properties.
- A **codec** is developed based on the format of data reported by devices. IoTDA uses codecs to convert data between binary and JSON formats as well as between different JSON formats. The binary data reported by a device is decoded into the JSON format for the application to read, and the commands delivered by the application are encoded into the binary or JSON format for the device to understand and execute. The following figure shows the process.

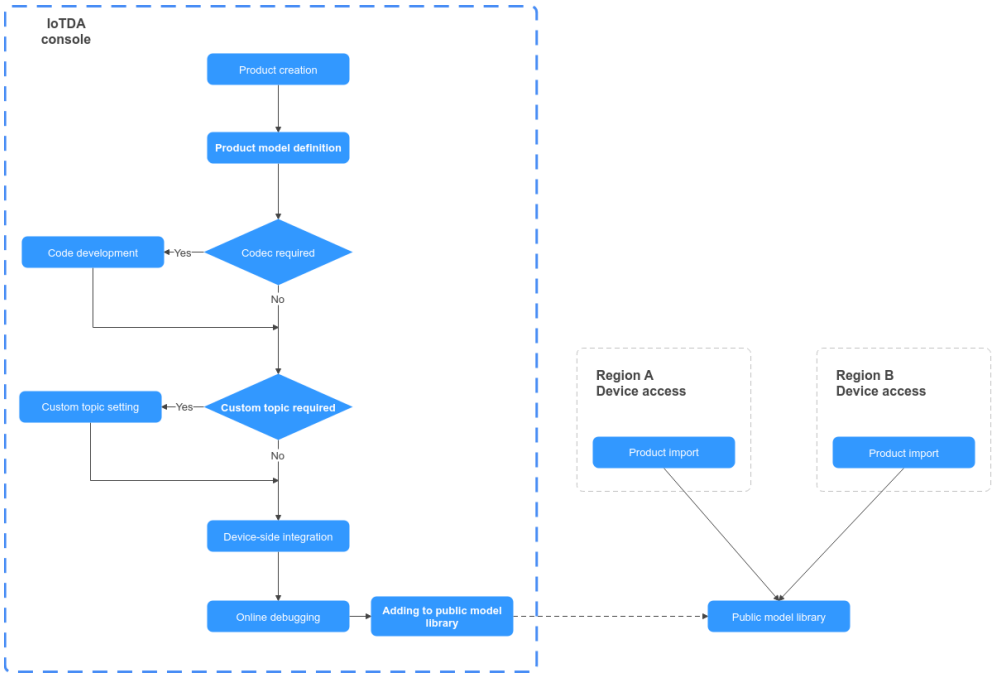
Figure 3-2 Codec usage process



Product Development Process

The IoTDA console provides a graphical user interface (GUI) to help you quickly develop products (product models and codecs) and perform self-service tests.

Figure 3-3 Product development process



- Product creation: A product is a collection of devices with the same capabilities or features. In addition to physical devices, a product includes product information, product models, and codecs generated during IoT capability building.
- Model definition: Product model development is the most important part of product development. A product model is used to describe the capabilities and features of a device. You can build an abstract model of a device by defining a product model on the platform so that the platform can know what services, properties, and commands are supported by the device.
- Codec development: If the data reported by the device is in binary or JSON format, a codec must be developed to convert data between binary and JSON formats or between different JSON formats.
- Online commissioning: IoTDA provides application and device simulators for you to commission data reporting and command delivery before developing real applications and physical devices. You can also use the application simulator to verify the service flow after the physical device is developed.

  NOTE

  Currently, only the standard edition supports online debugging of MQTT devices.

FAQ

Product Models

3.2.2 Creating a Product

On the IoT platform, a product is a collection of devices with the same capabilities or features.

Procedure

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 Choose **Products** in the navigation pane and click **Create Product** on the left. Set the parameters as prompted and click **OK**.

Set Basic Info	
Resource Space	Select a resource space from the drop-down list box. If a **resource space** does not exist, create it first.
Product Name	Define a product name. The product name must be unique in the same resource space. The value can contain up to 64 characters. Only letters, digits, and special characters (_?'#().,&%@!-) are allowed.
Protocol	• MQTT: The device data format can be binary or JSON. If the binary format is used, the codec must be deployed. • LwM2M over CoAP: Used only by NB-IoT devices with limited resources (including storage and power consumption). The data format is binary, requiring the codec for device-platform interaction. • HTTPS: A secure communication protocol based on HTTP and encrypted using SSL. • Modbus: Devices that access the platform with Modbus via IoT edge nodes (or child devices that connect to the platform through gateways) are indirectly connected devices. For details about the differences between directly connected devices and indirectly connected devices, see **Gateways and Child Devices**. • HTTP (TLS-encrypted), ONVIF, OPC UA, OPC DA, TCP, UDP, and other protocols: IoT Edge is used for connection.
Data Type	• **JSON**: JSON is used for the communication protocol between the platform and devices. • **Binary**: You need to develop a **codec** on the IoTDA console to convert binary code data reported by devices into JSON data. The devices can communicate with the platform only after the JSON data delivered by the platform is parsed into binary code.

Set Basic Info	
Encoding Format	When **protocol_type** is set to **MQTT** and **data_format** is set to binary, set this parameter to specify the encoding format of messages reported by devices. • **UTF-8** (default value): converts binary code streams into Unicode strings. • **BASE64**: converts binary code streams into Base64 strings.
Industry	Set this parameter based on service requirements.
Device Type	Set this parameter based on service requirements.
Advanced Settings	
Product ID	Set a unique identifier for the product. If this parameter is specified, the platform uses the specified product ID. If this parameter is not specified, the platform allocates a product ID.
Description	Provide a description for the product. Set this parameter based on service requirements.

You can click **More** > **Delete** to delete a product that is no longer used. After the product is deleted, its resources such as the product models and codecs will be cleared. Exercise caution when deleting a product.

----End

Follow-Up Procedure

1. In the product list, click the name of a product to access its details page. On the product details page displayed, you can view basic product information, such as the product ID, product name, device type, data format, resource space, and protocol type.

   Figure 3-4 Product details

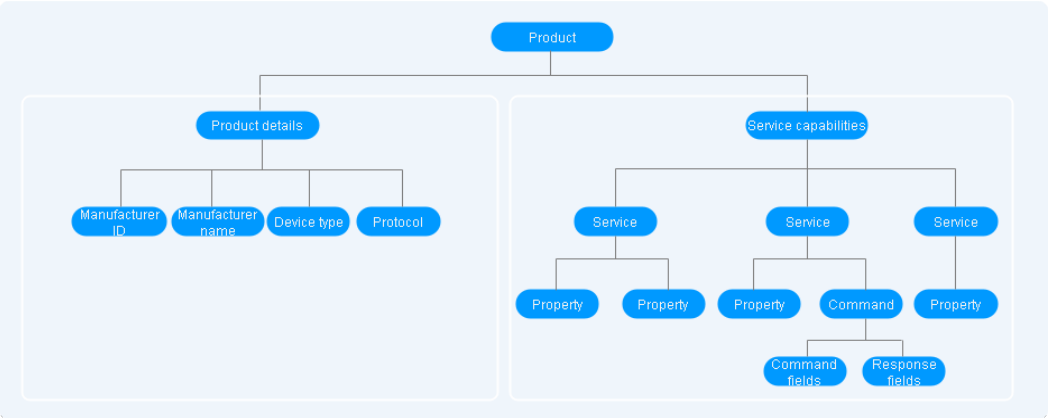
2. On the product details page, you can **develop a product model**, **develop a codec**, **perform online debugging**, and **customize topics**.

3.2.3 Developing a Product Model

3.2.3.1 Product Model Definition

A product model describes the capabilities and features of a device. You can build an abstract model of a device by defining a product model on the IoT platform so

that the platform can know what services, properties, and commands are supported by the device, such as its on/off switches. After defining a product model, you can use it during **device registration**.



A product model defines service capabilities.

- **Service capabilities**

  The service capabilities of a device are divided into several services. Properties, commands, and command parameters are defined for each service.

  For example, a water meter has multiple capabilities. It reports the water flow, alarms, battery life, and connection data, and it receives commands too. When describing the capabilities of a water meter, the product model includes five services, each of which has its own properties or commands.

Service Name	Description
WaterMeterBasic	Defines parameters reported by the water meter, such as the water flow, temperature, and pressure. If these parameters need to be controlled or modified using commands, these parameters must be defined in the commands.
WaterMeterAlarm	Defines various scenarios where the water meter will report an alarm. Commands need to be defined if necessary.
Battery	Defines the voltage and current intensity of the water meter.
DeliverySchedule	Defines transmission rules for the water meter. Commands need to be defined if necessary.
Connectivity	Defines connectivity parameters of the water meter.

NOTE

You can define the number of services as required. For example, the **WaterMeterAlarm** service can be further divided into **WaterPressureAlarm** and **WaterFlowAlarm** services or be integrated into the **WaterMeterBasic** service. The platform provides multiple methods for developing product models. You can select a method as required.

- **Customize Model (online development)**: Build a product model from scratch. For details, see **Developing a Product Model Online**.
- **Import from Local (offline development)**: Upload a local product model to the platform. For details, see **Developing a Product Model Offline**.
- **Import from Excel**: Define product functions by importing an Excel file. This method can lower the product model development threshold for developers because they only need to fill in parameters based on the Excel file. It also helps high-level developers and integrators improve the development efficiency of complex models in the industry. For example, the auto-control air conditioner model contains more than 100 service items. Developing the product model by editing the excel file greatly improves the efficiency. You can edit and adjust parameters at any time. For details, see **Import from Excel**.
- **Import from Library**: You can use a preset product model to quickly develop a product. The platform provides standard and manufacturer-specific product models. Standard product models comply with industry standards and are suitable for devices of most manufacturers in the industry. Manufacturer-specific product models are suitable for devices provided by a small number of manufacturers. You can select a product model as required.

3.2.3.2 Developing a Product Model Online

Overview

Before developing a product model online, you must **create a product**. When creating a product, enter information such as the product name, protocol type, data format, industry, and device type. The information will be used to fill in the device capability fields in the product model. The IoT platform provides standard models and vendor models. These models involve multiple domains and provide edited product model files. You can modify, add, or delete fields in the product model as required. If you want to customize a product model, you need to define a complete product model.

This topic uses a product model that contains a service as an example. The product model contains functions and fields in scenarios such as data reporting, command delivery, and command response delivery.

Procedure

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Products**. In the product list, click the name of a product to access its details.

Step 3 On the **Basic Information** tab page, click the button for adding a service.

Step 4 Specify **Service ID**, **Service Type**, and **Description**, and click **OK**.

- **Service ID**: The first letter of the value must be capitalized, for example, WaterMeter and StreetLight.
- **Service Type**: You are advised to set this parameter to the same value as **Service ID**.

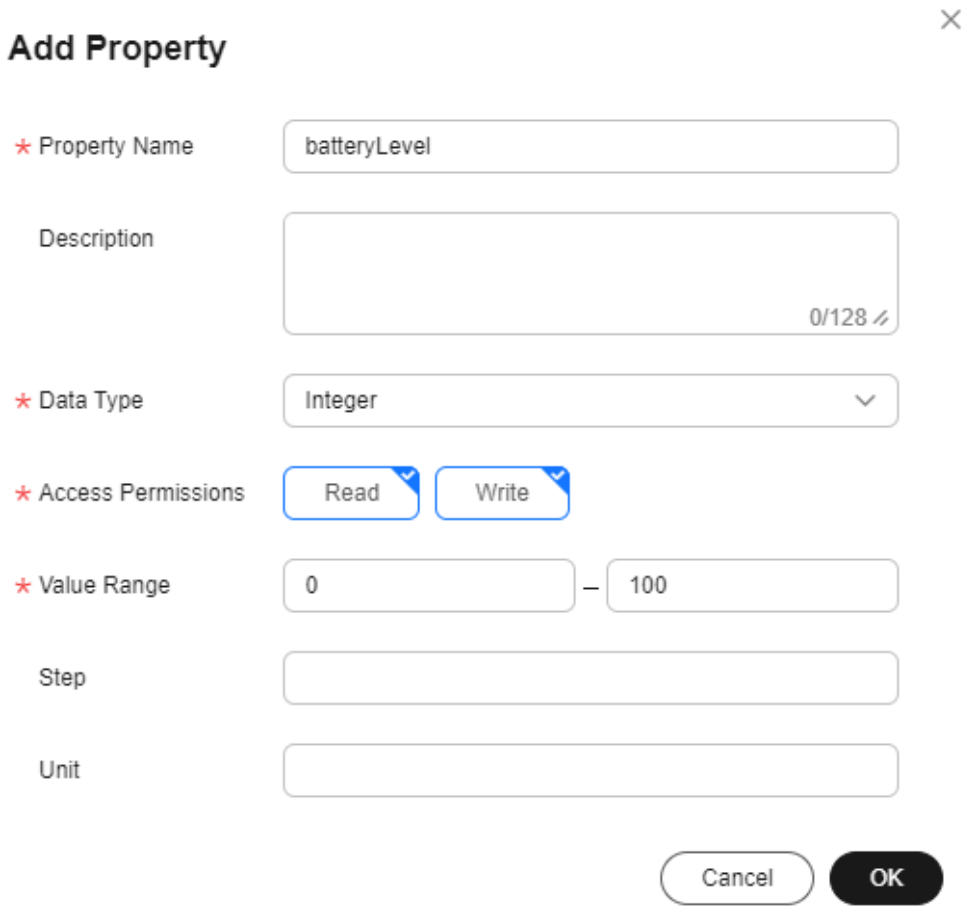
- **Description**: You can, for example, define the properties of light intensity (Light_Intensity) and status (Light_Status).

After the service is added, define the properties and commands in the **Add Service** area. A service can contain properties and/or commands. Configure the properties and commands based on your requirements.

Step 5 Click the new service ID added in **4**. On the page displayed, click **Add Property**. In the dialog box displayed, set the parameters and click **OK**.

Parameter	Description
Property Name	Use camel case, for example, **batteryLevel** and **internalTemperature**.
Data Type	• **Integer**: Select this value if the reported data is an integer value. • **long**: Select this value if the reported data is a long integer. • **Decimal**: Select this value if the reported data is a decimal. You are advised to set this parameter to **Decimal** when configuring the longitude and latitude properties. • **String**: Select this value if the reported data is a string or an enumerated value. Use commas (,) to separate values. • **DateTime**: Select this value if the reported data is a date or time. Property format examples: **2020-09-01T18:50:20Z** and **2020-09-01T18:50:20.200Z** • **JsonObject**: Select this value if the reported data is in JSON structure. • **enum**: Select this value if the reported data is enumerated values. If enumerated values are **OPEN,CLOSE**, property format examples include **OPEN** and **CLOSE**. • **boolean**: Select this value if the reported data is a Boolean value. Property format examples: **true/false** and **0/1** • **StringList**: Select this value if the reported data is a string list. Property format examples: **["str1","str2","str3"]**
Access Permissions	• **Read**: You can query the property through APIs. • **Write**: You can modify the property value through APIs.
Value Range	Set these parameters based on the actual situation of the device.
Step	
Unit	

Figure 3-5 Adding a property - batteryLevel

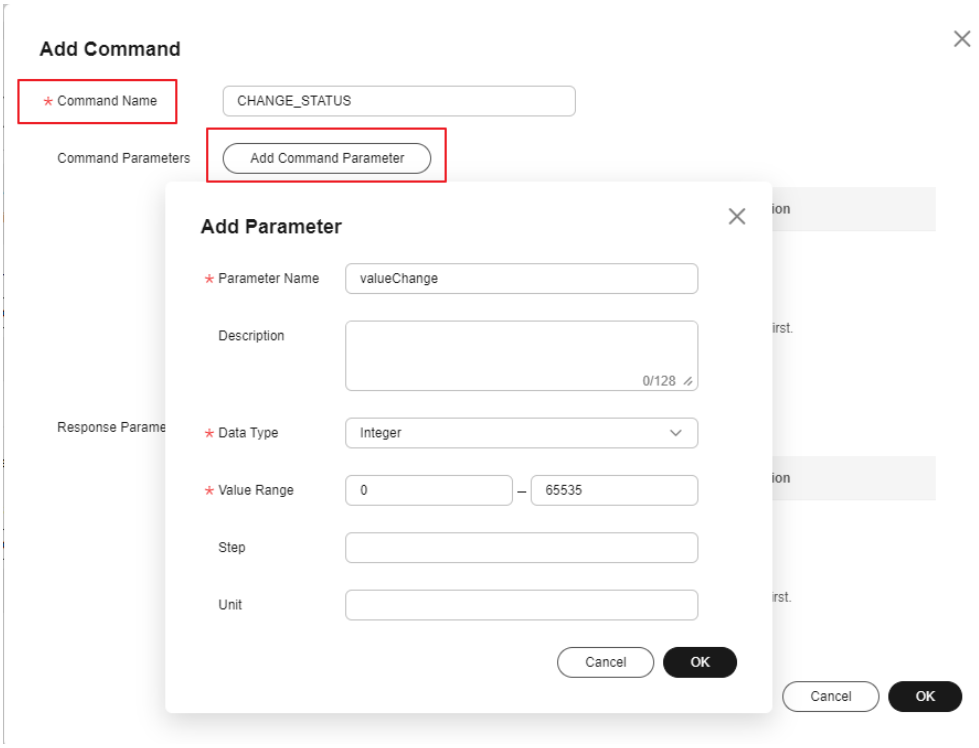


Step 6 Click **Add Command**. In the dialog box displayed, set command parameters.

- **Command Name**: You are advised to capitalize the full command name and use underscores (_) to separate words, for example, **DISCOVERY** and **CHANGE_STATUS**.
- **Command Parameters**: Click **Add Command Parameter**. In the dialog box displayed, set the parameters of the command to be delivered and click **OK**.

Parameter	Description
Parameter	You are advised to start the name with a lowercase letter and capitalize the other words, example, **valueChange**.
Data Type	Set these parameters based on the actual situation of the device.
Value Range	
Step	
Unit	

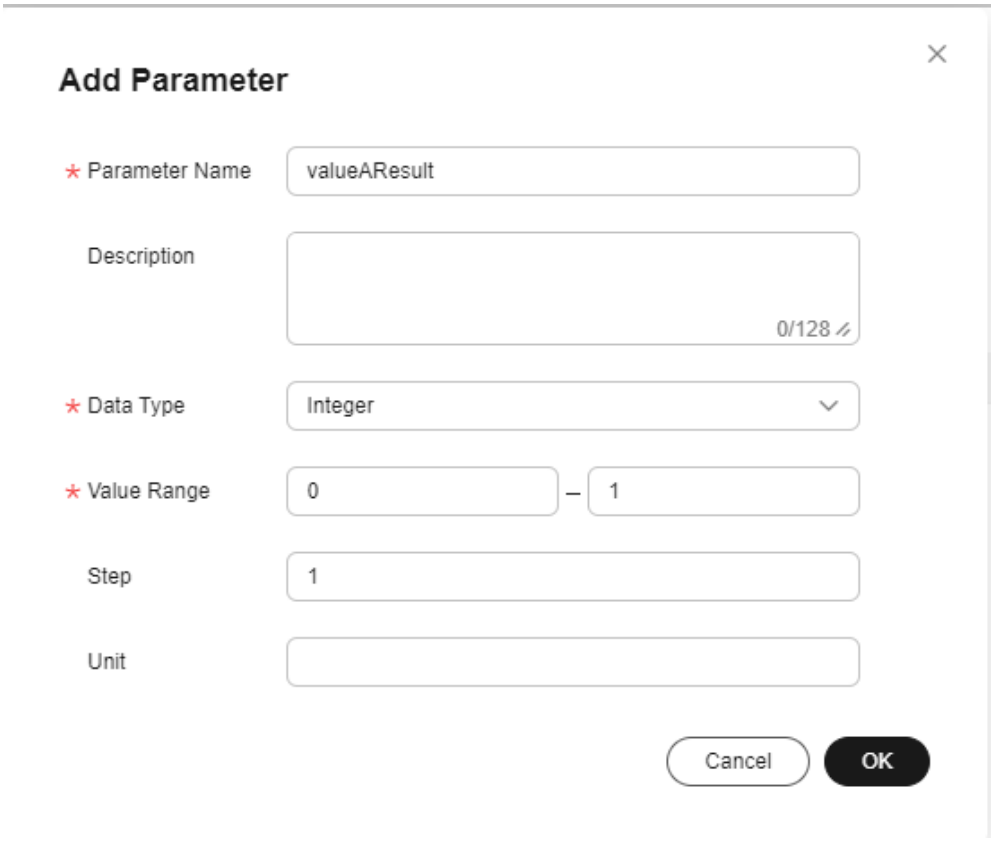
Figure 3-6 Adding a command - CHANGE_STATUS



- Click **Add Response Parameter** to add parameters of a command response when necessary. In the dialog box displayed, set the parameters and click **OK**.

Parameter	Description
Parameter	You are advised to start the name with a lowercase letter and capitalize the other words, example, **valueResult**.
Data Type	Set these parameters based on the actual situation of the device.
Value Range	
Step	
Unit	

Figure 3-7 Adding a command response parameter - valueAResult

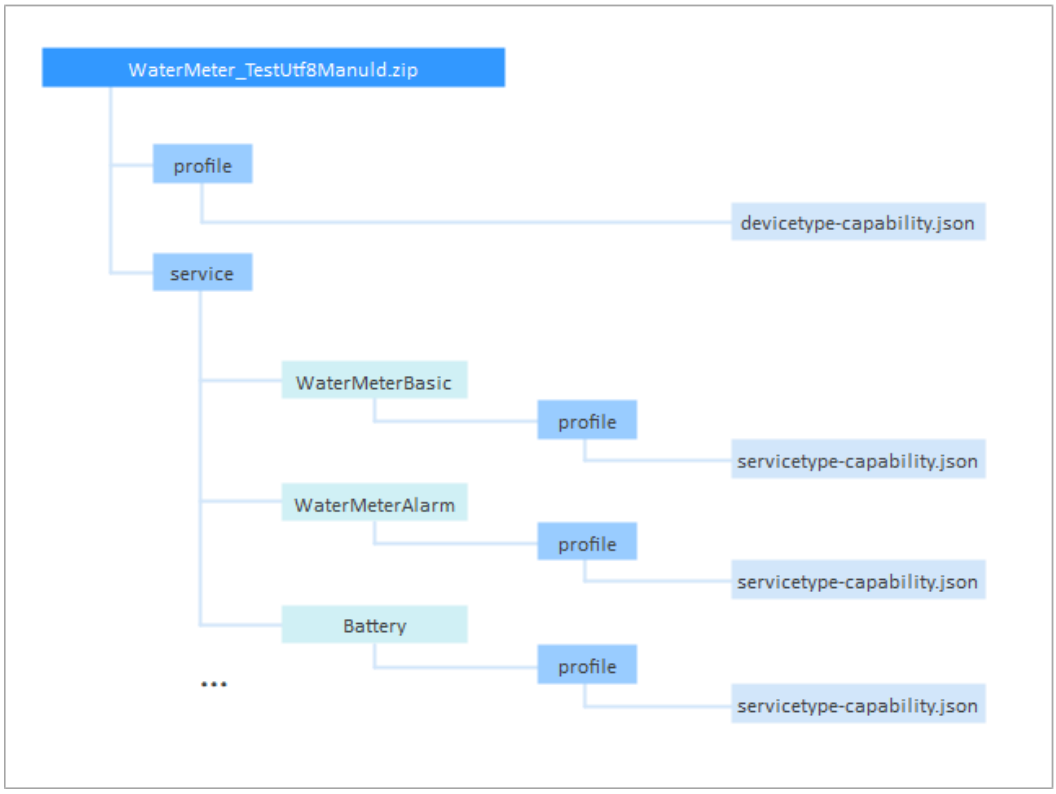


----End

3.2.3.3 Developing a Product Model Offline

Overview

A product model is essentially a .zip package consisting of a **devicetype-capability.json** file and several **serviceType-capability.json** files. The **devicetype-capability.json** file describes the service capabilities contained in the product model, and the **serviceType-capability.json** file describes each capability of **service_capabilities** in the **devicetype-capability.json** file. **WaterMeter** indicates the device type, **TestUtf8Manuld** identifies the manufacturer ID, and **WaterMeterBasic**, **WaterMeterAlarm**, and **Battery** indicates the service types.



In offline development, you need to define device capabilities in the **devicetype-capability.json** file and service capabilities in the **servicetype-capability.json** file based on the platform rules and JSON format specifications.

Developing a Product Model Online is recommended, which is less time-consuming.

Naming Rules

The product model must comply with the following naming rules:

- Use upper camel case for device types, service types, and service IDs, for example, **WaterMeter** and **Battery**.
- Use lower camel case for property names, for example, **batteryLevel** and **internalTemperature**.
- For commands, capitalize all characters, with words separated by underscores, for example, **DISCOVERY** and **CHANGE_COLOR**.
- Name a device capability profile (.json file) in the format of **devicetype-capability.json**.
- Name a service capability profile (.json file) in the format of **servicetype-capability.json**.
- The manufacturer ID must be unique in different product models and can only be in English.
- Names are universal and concise and service capability descriptions clearly indicate corresponding functions. For example, you can name a multi-sensor device **MultiSensor** and name a service that displays the battery level **Battery**.

Product Model Templates

To connect a new device to the IoT platform, you must first define a product model for the device. The IoT platform provides some product model templates. If the types and functions of devices newly connected to the IoT platform are included in these templates, directly use the templates. If the types and functions are not included in the product model templates, define your product model.

For example, if a water meter is connected to the IoT platform, you can directly select the corresponding product model on the IoT platform and modify the device service list.

NOTE

The product model templates provided by the IoT platform are updated continuously. The following uses a water meter as an example to describe how to define a product model.

Device identification properties

Property	Key (Product Model JSON File)	Value
Device Type	deviceType	WaterMeter
Manufacturer ID	manufacturerId	TestUtf8ManuId
Manufacturer Name	manufacturerName	HZYB
Protocol Type	protocolType	CoAP

Service list

Service	Service ID	Service Type	Option
Basic water meter function	WaterMeterBasic	Water	Mandatory
Alarm service	WaterMeterAlarm	Battery	Mandatory
Battery service	Battery	Battery	Optional
Data reporting rule	DeliverySchedule	DeliverySchedule	Mandatory
Connectivity	Connectivity	Connectivity	Mandatory

Device Capability Definition Example

The **devicetype-capability.json** file records basic information about a device.

```
{
   "devices": [
      {
         "manufacturerId": "TestUtf8ManuId",
         "manufacturerName": "HZYB",
```



```
        "protocolType": "CoAP",
        "deviceType": "WaterMeter",
        "omCapability":{
                    "upgradeCapability" : {
                        "supportUpgrade":true,
                        "upgradeProtocolType":"PCP"
                    },
                    "fwUpgradeCapability" : {
                        "supportUpgrade":true,
                        "upgradeProtocolType":"LWM2M"
                    },
                    "configCapability" : {
                        "supportConfig":true,
                        "configMethod":"file",
                        "defaultConfigFile": {
                          "waterMeterInfo" : {
                              "waterMeterPirTime" : "300"
                           }
                        }
                    }
        },
        "serviceTypeCapabilities": [
          {
             "serviceId": "WaterMeterBasic",
             "serviceType": "WaterMeterBasic",
             "option": "Mandatory"
          },
          {
             "serviceId": "WaterMeterAlarm",
             "serviceType": "WaterMeterAlarm",
             "option": "Mandatory"
          },
          {
             "serviceId": "Battery",
             "serviceType": "Battery",
             "option": "Optional"
          },
          {
             "serviceId": "DeliverySchedule",
             "serviceType": "DeliverySchedule",
             "option": "Mandatory"
          },
          {
             "serviceId": "Connectivity",
             "serviceType": "Connectivity",
             "option": "Mandatory"
          }
        ]
    }
  ]
}
```

The fields are described as follows:

Field	Sub-field		Mandatory	Description
devices	-	-	Yes	Complete capability information about a device. The root node cannot be modified.
-	manufacturerId	-	No	Manufacturer ID of the device.
-	manufacturerName	-	Yes	Manufacturer name of the device. The name must be in English.

Field	Sub-field		Mandatory	Description
-	protocolType	-	Yes	Protocol used by the device to connect to the IoT platform. For example, the value is **CoAP** for NB-IoT devices.
-	deviceType	-	Yes	Type of the device.
-	omCapability	-	No	Software upgrade, firmware upgrade, and configuration update capabilities of the device. For details, see the description of the omCapability structure below. If software or firmware upgrade is not involved, this field can be deleted.
-	serviceTypeCapabilities	-	Yes	Service capabilities of the device.
-	-	serviceId	Yes	Service ID. If a service type includes only one service, the value of **serviceId** is the same as that of **serviceType**. If the service type includes multiple services, the services are numbered correspondingly, such as Switch01, Switch02, and Switch03.
-	-	serviceType	Yes	Type of the service. The value of this field must be the same as that of **serviceType** in the **servicetype-capability.json** file.
-	-	option	Yes	Type of the service field. The value can be **Master**, **Mandatory**, or **Optional**. This field is not a functional field but a descriptive one.

Description of the omCapability structure

Field	Sub-field	Mandatory	Description
upgradeCapability	-	No	Software upgrade capabilities of the device.

Field	Sub-field	Mandatory	Description
-	supportUpgrade	No	**true**: The device supports software upgrades. **false**: The device does not support software upgrades.
-	upgradeProtocolType	No	Protocol type used by the device for software upgrades. It is different from **protocolType** of the device. For example, the software upgrade protocol of CoAP devices is PCP.
fwUpgradeCapability	-	No	Firmware upgrade capabilities of the device.
-	supportUpgrade	No	**true**: The device supports firmware upgrades. **false**: The device does not support firmware upgrades.
-	upgradeProtocolType	No	Protocol type used by the device for firmware upgrades. It is different from **protocolType** of the device. Currently, the IoT platform supports only firmware upgrades of LwM2M devices.
configCapability	-	No	Configuration update capabilities of the device.
-	supportConfig	No	**true**: The device supports configuration updates. **false**: The device does not support configuration updates.
-	configMethod	No	**file**: Configuration updates are delivered in the form of files.
-	defaultConfigFile	No	Default device configuration information (in JSON format). The specific configuration information is defined by the manufacturer. The IoT platform stores the information for delivery but does not parse the configuration fields.

Service Capability Definition Example

The **servicetype-capability.json** file records service information about a device.

```
{
   "services": [
      {
```



```
"serviceType": "WaterMeterBasic",
"description": "WaterMeterBasic",
"commands": [
   {
      "commandName": "SET_PRESSURE_READ_PERIOD",
      "paras": [
         {
            "paraName": "value",
            "dataType": "int",
            "required": true,
            "min": 1,
            "max": 24,
            "step": 1,
            "maxLength": 10,
            "unit": "hour",
            "enumList": null
         }
      ],
      "responses": [
         {
            "responseName": "SET_PRESSURE_READ_PERIOD_RSP",
            "paras": [
               {
                  "paraName": "result",
                  "dataType": "int",
                  "required": true,
                  "min": -1000000,
                  "max": 1000000,
                  "step": 1,
                  "maxLength": 10,
                  "unit": null,
                  "enumList": null
               }
            ]
         }
      ]
   }
],
"properties": [
   {
      "propertyName": "registerFlow",
      "dataType": "int",
      "required": true,
      "min": 0,
      "max": 0,
      "step": 1,
      "maxLength": 0,
      "method": "R",
      "unit": null,
      "enumList": null
   },
   {
      "propertyName": "currentReading",
      "dataType": "string",
      "required": false,
      "min": 0,
      "max": 0,
      "step": 1,
      "maxLength": 0,
      "method": "W",
      "unit": "L",
      "enumList": null
   },
   {
      "propertyName": "timeOfReading",
      "dataType": "string",
      "required": false,
      "min": 0,
      "max": 0,
```

```
                "step": 1,
                "maxLength": 0,
                "method": "W",
                "unit": null,
                "enumList": null
            },
            ......
        ]
    }
  ]
}
```

The fields are described as follows:

Field	Sub-field				Mandatory	Description
services	-	-	-	-	Yes	Complete information about a service. The root node cannot be modified.
-	serviceType	-	-	-	Yes	Type of the service. The value of this field must be the same as that of **serviceType** in the **devicetype-capability.json** file.
-	description	-	-	-	Yes	Description of the service. This field is not a functional field but a descriptive one. It can be set to **null**.
-	commands	-	-	-	Yes	Command supported by the device. If the service has no commands, set the value to **null**.
-	-	commandName	-	-	Yes	Name of the command. The command name and parameters together form a complete command.
-	-	paras	-	-	Yes	Parameters contained in the command.
-	-	-	paraName	-	Yes	Name of a parameter in the command.

Field	Sub-field				Mandatory	Description
-	-	-	dataType	-	Yes	Data type of the parameter in the command. Value: **string**, **int**, **string list**, **decimal**, **DateTime**, **jsonObject**, **enum**, or **boolean** Complex types of reported data are as follows: • string list:["str1","str2","str3"] • **DateTime**: The value is in the format of yyyyMMdd'T'HHmmss'Z', for example, 20151212T121212Z. • **jsonObject**: The value is in the customized JSON format, which is not parsed by the IoT platform and is transparently transmitted only.
-	-	-	required	-	Yes	Whether the command is mandatory. The value can be **true** or **false**. The default value is **false**, indicating that the command is optional. This field is not a functional field but a descriptive one.
-	-	-	min	-	Yes	Minimum value. This field is valid only when **dataType** is set to **int** or **decimal**.
-	-	-	max	-	Yes	Maximum value. This field is valid only when **dataType** is set to **int** or **decimal**.
-	-	-	step	-	Yes	Step. This field is not used. Set it to **0**.
-	-	-	maxLength	-	Yes	String length. This field is valid only when **dataType** is set to **string**, **string list**, or **DateTime**.
-	-	-	unit	-	Yes	Unit, which must be in English. The value is determined by the parameter, for example: Temperature unit: C or K Percentage unit: % Pressure unit: Pa or kPa

Field	Sub-field				Mandatory	Description
-	-	-	enumList	-	Yes	Enumerated value. For example, the status of a switch can be set as follows: "enumList" : ["OPEN","CLOSE"] This field is not a functional field but a descriptive one. It is recommended that this field be defined accurately.
-	-	responses	-	-	Yes	Responses to command execution.
-	-	-	responseName	-	Yes	You can add _RSP to the end of **commandName**.
-	-	-	paras	-	Yes	Parameters contained in a response.
-	-	-	-	paraName	Yes	Name of a parameter in the command.
-	-	-	-	dataType	Yes	Data type. Value: **string**, **string list**, **decimal**, **DateTime**, **jsonObject**, or **int** Complex types of reported data are as follows: ● string list:["str1","str2","str3"] ● **DateTime**: The value is in the format of yyyyMMdd'T'HHmmss'Z', for example, 20151212T121212Z. ● **jsonObject**: The value is in the customized JSON format, which is not parsed by the IoT platform and is transparently transmitted only.
-	-	-	-	required	Yes	Whether the command response is mandatory. The value can be **true** or **false**. The default value is **false**, indicating that the command response is optional. This field is not a functional field but a descriptive one.

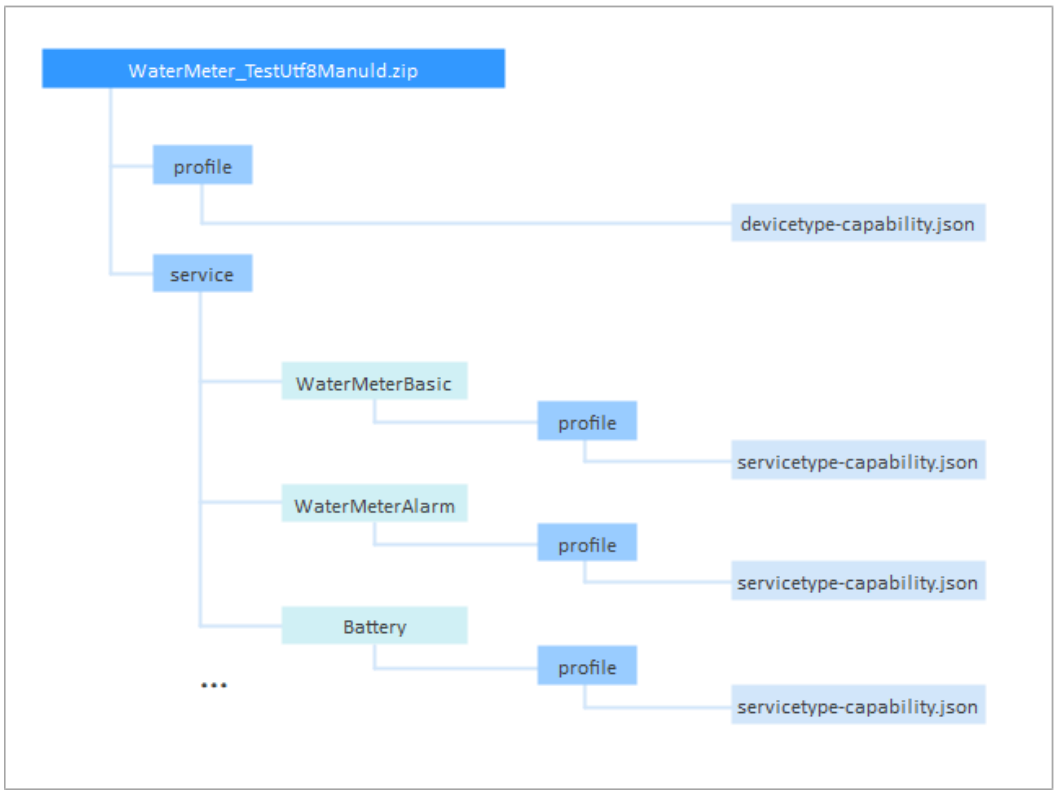
Field	Sub-field				Mandatory	Description
-	-	-	-	min	Yes	Minimum value. This field is valid only when **dataType** is set to **int** or **decimal**. The value of a field of the **int** or **decimal** type must be greater than or equal to the value of **min**.
-	-	-	-	max	Yes	Maximum value. This field is valid only when **dataType** is set to **int** or **decimal**. The value of a field of the **int** or **decimal** type must be less than or equal to the value of **max**.
-	-	-	-	step	Yes	Step. This field is not used. Set it to **0**.
-	-	-	-	maxLength	Yes	String length. This field is valid only when **dataType** is set to **string**, **string list**, or **DateTime**.
-	-	-	-	unit	Yes	Unit, which must be in English. The value is determined by the parameter, for example: Temperature unit: C or K Percentage unit: % Pressure unit: Pa or kPa
-	-	-	-	enumList	Yes	Enumerated value. For example, the status of a switch can be set as follows: "enumList" : ["OPEN","CLOSE"] This field is not a functional field but a descriptive one. It is recommended that this field be defined accurately.
-	properties	-	-	-	Yes	Reported data. Each sub-node indicates a property.
-	-	propertyName	-	-	Yes	Property name.

Field	Sub-field				Mandatory	Description
-	-	dataType	-	-	Yes	Data type. Value: **string**, **string list**, **decimal**, **DateTime**, **jsonObject**, or **int** Complex types of reported data are as follows: • string list:["str1","str2","str3"] • **DateTime**: The value is in the format of yyyyMMdd'T'HHmmss'Z', for example, 20151212T121212Z. • **jsonObject**: The value is in the customized JSON format, which is not parsed by the IoT platform and is transparently transmitted only.
-	-	required	-	-	Yes	Whether the property is mandatory. The value can be **true** or **false**. The default value is **false**, indicating that the property is optional. This field is not a functional field but a descriptive one.
-	-	min	-	-	Yes	Minimum value. This field is valid only when **dataType** is set to **int** or **decimal**. The value of a field of the **int** or **decimal** type must be greater than or equal to the value of **min**.
-	-	max	-	-	Yes	Maximum value. This field is valid only when **dataType** is set to **int** or **decimal**. The value of a field of the **int** or **decimal** type must be less than or equal to the value of **max**.
-	-	step	-	-	Yes	Step. This field is not used. Set it to **0**.
-	-	method	-	-	Yes	Access mode. **R** indicates reading, **W** indicates writing, and **E** indicates subscription. Value: R, RW, RE, RWE, or null

Field	Sub-field				Mandatory	Description
-	-	unit	-	-	Yes	Unit, which must be in English. The value is determined by the parameter, for example: Temperature unit: C or K Percentage unit: % Pressure unit: Pa or kPa
-	-	maxLength	-	-	Yes	String length. This field is valid only when **dataType** is set to **string**, **string list**, or **DateTime**.
-	-	enumList	-	-	Yes	Enumerated value. For example, batteryStatus can be set as follows: "enumList" : [0, 1, 2, 3, 4, 5, 6] This field is not a functional field but a descriptive one. It is recommended that this field be defined accurately.

Product Model Packaging

After the product model is completed, package it in the format shown below.



The following requirements must be met for product model packaging:

- The product model hierarchy must be the same as that shown above and cannot be added or deleted. For example, the second level can contain only the **profile** and **service** folders, and each service must contain the **profile** folder.
- The product model is compressed in **.zip** format.
- The product model must be named in the format of deviceType_manufacturerId. The values of **deviceType** and **manufacturerId** must be the same as those in the **devicetype-capability.json** file. For example, the following provides the main fields of the **devicetype-capability.json** file.

```
{
   "devices": [
      {
         "manufacturerId": "TestUtf8ManuId",
         "manufacturerName": "HZYB",

         "protocolType": "CoAP",
         "deviceType": "WaterMeter",
         "serviceTypeCapabilities": ****
      }
   ]
}
```

- WaterMeterBasic, WaterMeterAlarm, and Battery in the figure are services defined in the **devicetype-capability.json** file.

The product model is in the JSON format. After the product model is edited, you can use format verification websites on the Internet to check the validity of the JSON file.

3.2.3.4 Exporting and Importing a Product Model

A product model can be exported from or imported to the IoT platform.

- After a product is developed, tested, and verified, you can export the online defined product model to the local host.
- If you have a complete product model (developed offline or exported from other projects or platforms) or use an Excel file to develop a product model, you can import the product model to the platform.

Exporting a Product Model

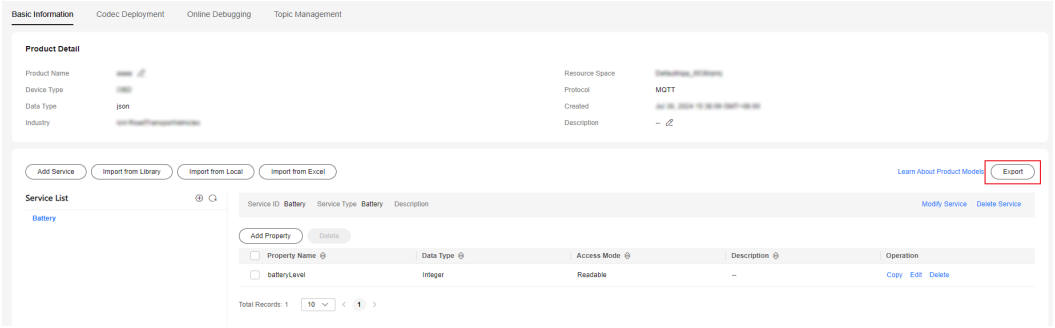
After a product is developed, tested, and verified, you can export the online defined product model to the local host.

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Products**. In the product list, click the name of a product to access its details.

Step 3 On the page displayed, click **Export** to export the product model to the local host.

Figure 3-8 Product Model - Exporting a product model



----End

Importing a Product Model

If you have a complete product model (developed offline or exported from other projects or platforms) or use an Excel file to develop a product model, you can import the product model to the platform.

NOTE

IoTDA uses codecs to convert data between binary and JSON formats as well as between JSON formats (see **Codec Definition**). The product model imported from the local host does not contain a codec. If the device reports binary code, go to the IoTDA console to develop or import a codec.

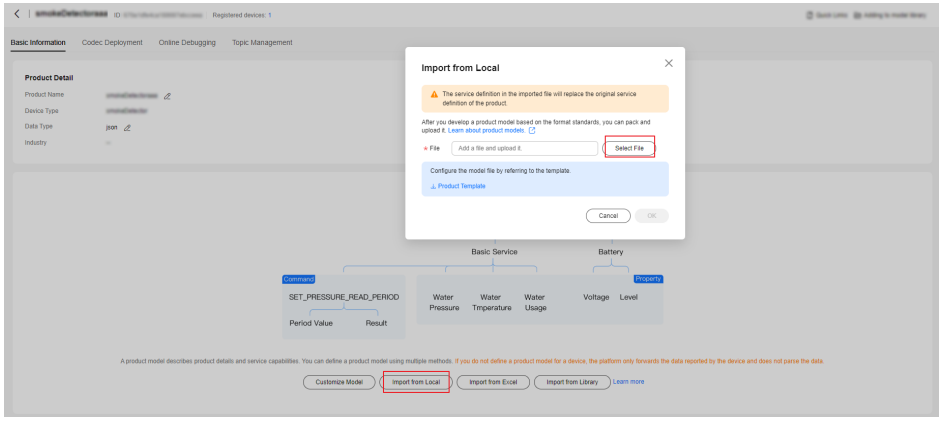
- **Import from Local**

  a. Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

  b. In the navigation pane, choose **Products**. In the product list, click the name of a product to access its details.

  c. On the **Basic Information** tab page, click **Import from Local**. In the dialog box displayed, load the local product model and click **OK**.

  Figure 3-9 Product - Uploading a product model

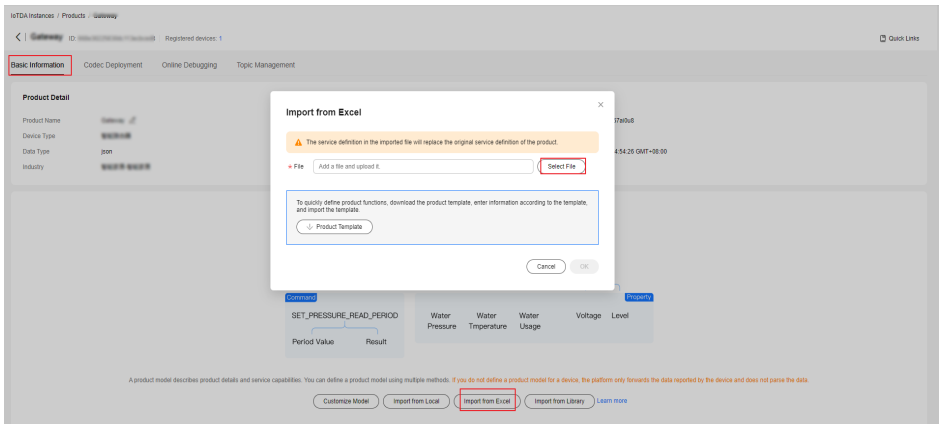
- **Import from Excel**

  a. Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

b. In the navigation pane, choose **Products**. In the product list, click the name of a product to access its details.

c. On the **Model Definition** tab page, click **Import from Excel**. In the product template downloaded, enter the service ID in the **Device** sheet and set parameters such as properties, commands, and events in the **Parameter** sheet. Import the Excel file and click **OK**.

Figure 3-10 Product - Importing a product model using an Excel file

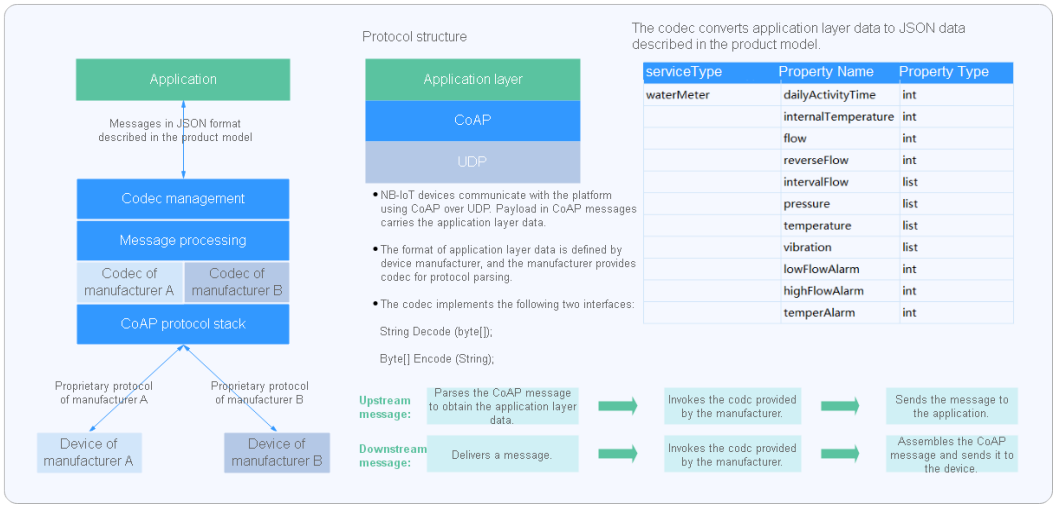


3.2.4 Developing a Codec

3.2.4.1 Codec Definition

A codec, as a plug-in within the platform, enables the conversion of data between binary and JSON formats or between different JSON formats. It manages the conversion of data from devices to the platform and vice versa.

In the NB-IoT scenario, a codec can decode binary data reported by a device into the JSON format for the application to read, and encode the commands delivered by the application into the binary format for the device to understand and execute. CoAP is used for communications between NB-IoT devices and the IoT platform. The payload of CoAP messages carries data at the application layer, at which the data type is defined by the devices. As NB-IoT devices require low power consumption, data at the application layer is generally in binary format instead of JSON. However, the platform sends data in JSON format to applications. Therefore, codec development is required for the platform to convert data between binary and JSON formats.



Scenarios

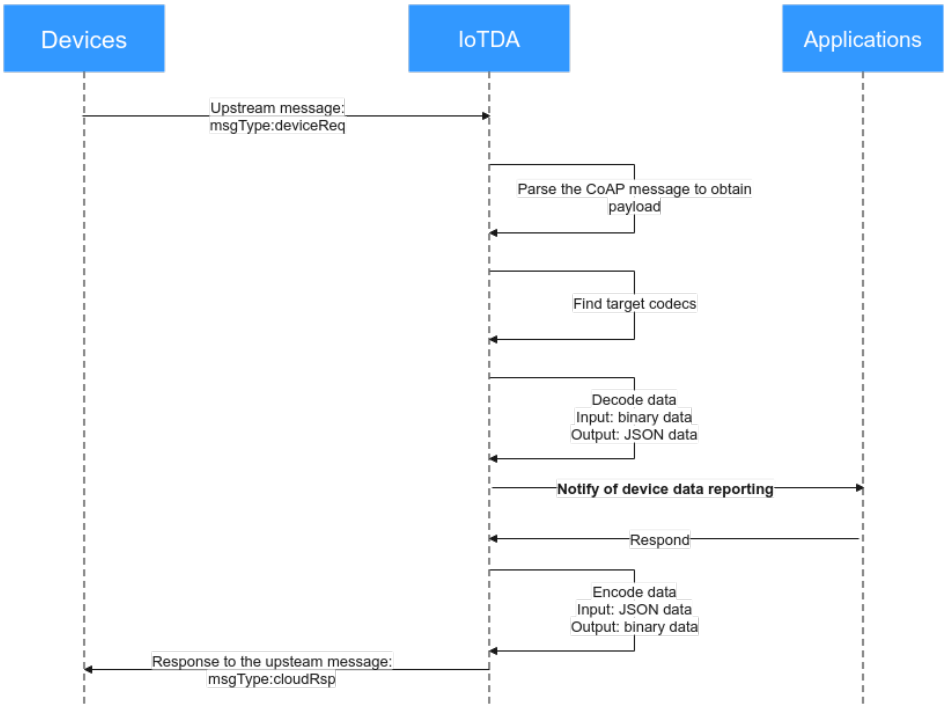
1. Required for device access using LwM2M/CoAP

2. Required for device access using generic protocols, such as TCP, JT808, and GB32960

3. Not required for device access using MQTT(S) or HTTP(S)

Data Reporting

Figure 3-11 Codecs for data reporting

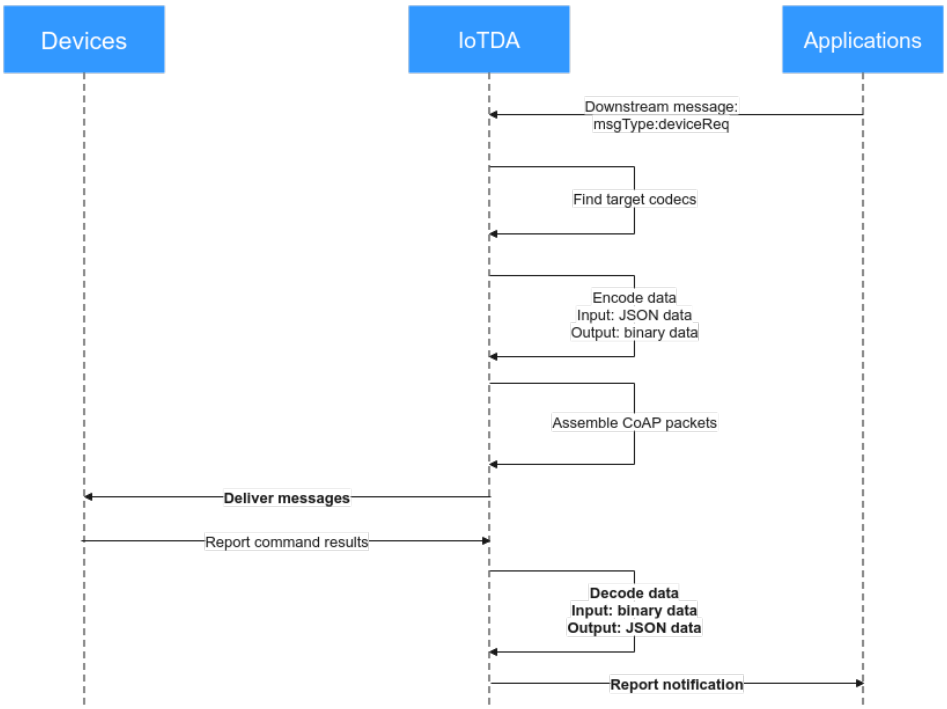


In the data reporting process, the codec is used in the following scenarios:

- Decoding binary data reported by a device into JSON data and sending the decoded data to an application
- Encoding JSON data returned by an application into binary data that can be identified by the device and sending the encoded data to a device

Command Delivery

Figure 3-12 Codec usage in command delivery



In the command delivery process, the codec is used in the following scenarios:

- Encoding JSON data delivered by an application into binary data and sending the encoded data to a device
- Decoding binary data returned by a device into JSON data and reporting the decoded data to an application

Graphical Development and Script-based Development

The platform provides multiple methods for developing codecs.

- **Online development**: The codec of a product can be quickly developed in a visualized manner on the IoTDA console.
- **Script-based development**: JavaScript scripts are used to implement encoding and decoding. After December 1, 2024, JavaScript-based codec development is no longer available on the platform for new users. You are advised to use FunctionGraph to write JavaScript scripts. For details, see **Overview**.

FAQ

Codecs

What Is NB-IoT?

Best Practices

Connecting and Debugging an NB-IoT Smart Street Light Using a Simulator

3.2.4.2 Online Development

Codecs developed online on IoTDA apply only to devices that report binary data.

On the IoTDA console, you can quickly develop codecs in a visualized manner.

This section uses an NB-IoT smoke detector as an example to describe how to develop a codec that supports data reporting and command delivery as well as command execution result reporting. The other two scenarios are used as examples to describe how to develop and commission complex codecs.

- **Codec for Data Reporting and Command Delivery**
- **Codec for Strings and Variable-Length Strings**
- **Codec for Arrays and Variable-Length Arrays**

Codec for Data Reporting and Command Delivery

Scenario

A smoke detector provides the following functions:

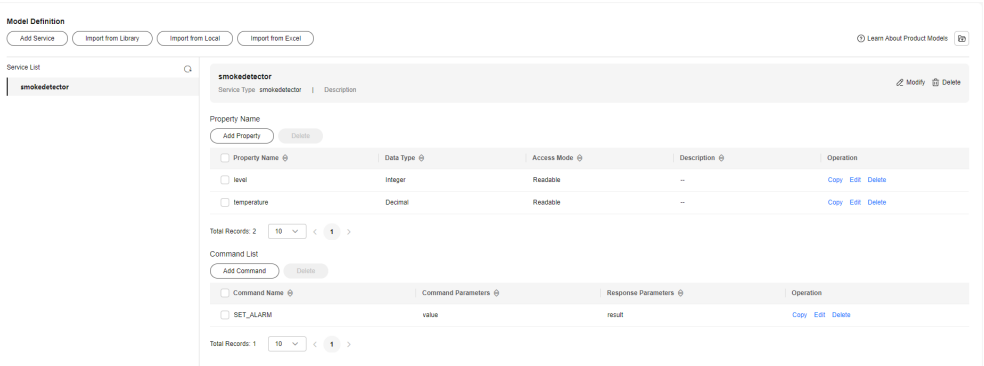
- Reporting smoke alarms (fire severity) and temperature.
- Receiving and running remote control commands, which can be used to enable the alarm function remotely. For example, the smoke detector can report the temperature on the fire scene and remotely trigger a smoke alarm for evacuation.
- Reporting command execution results

Product Model

Define the **product model** on the product details page of the smoke detector.

- **level**: indicates the fire severity.
- **temperature**: indicates the temperature at the fire scene.
- **SET_ALARM**: indicates whether to enable or disable the alarm function. The value **0** indicates that the alarm function is disabled, and **1** indicates that the alarm function is enabled.

Figure 3-13 Model definition - smokedetector



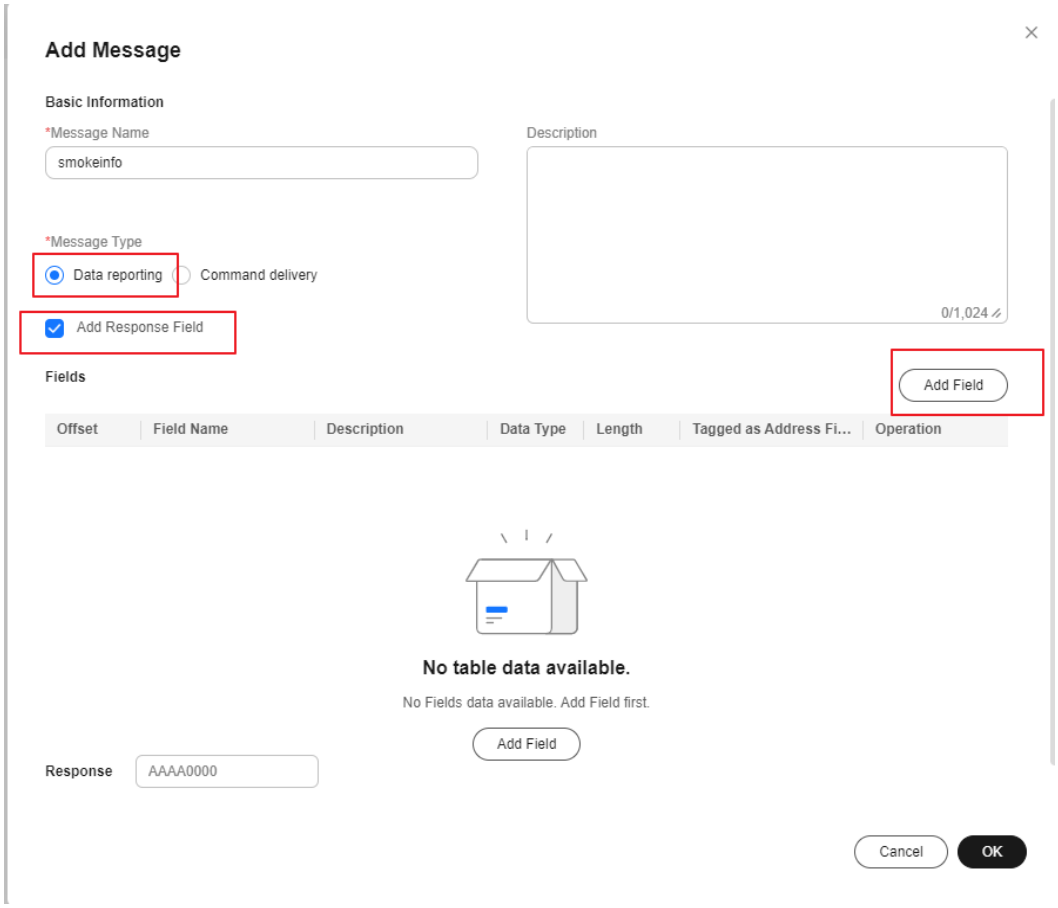
Developing a Codec

Step 1 On the smoke detector details page, click the **Codec Development** tab and click **Develop Codec**.

Step 2 Click **Add Message** to add a **smokerinfo** message. This step is performed to decode the binary code stream message uploaded by the device to the JSON format so that the platform can understand the message. The following is a configuration example:

- **Message Name**: smokerinfo
- **Message Type**: Data reporting
- **Add Response Field**: selected. After response fields are added, the platform delivers the response data set by the application to the device after receiving the data reported by the device.
- **Response**: AAAA0000 (default)

Figure 3-14 Adding a message - smokerinfo

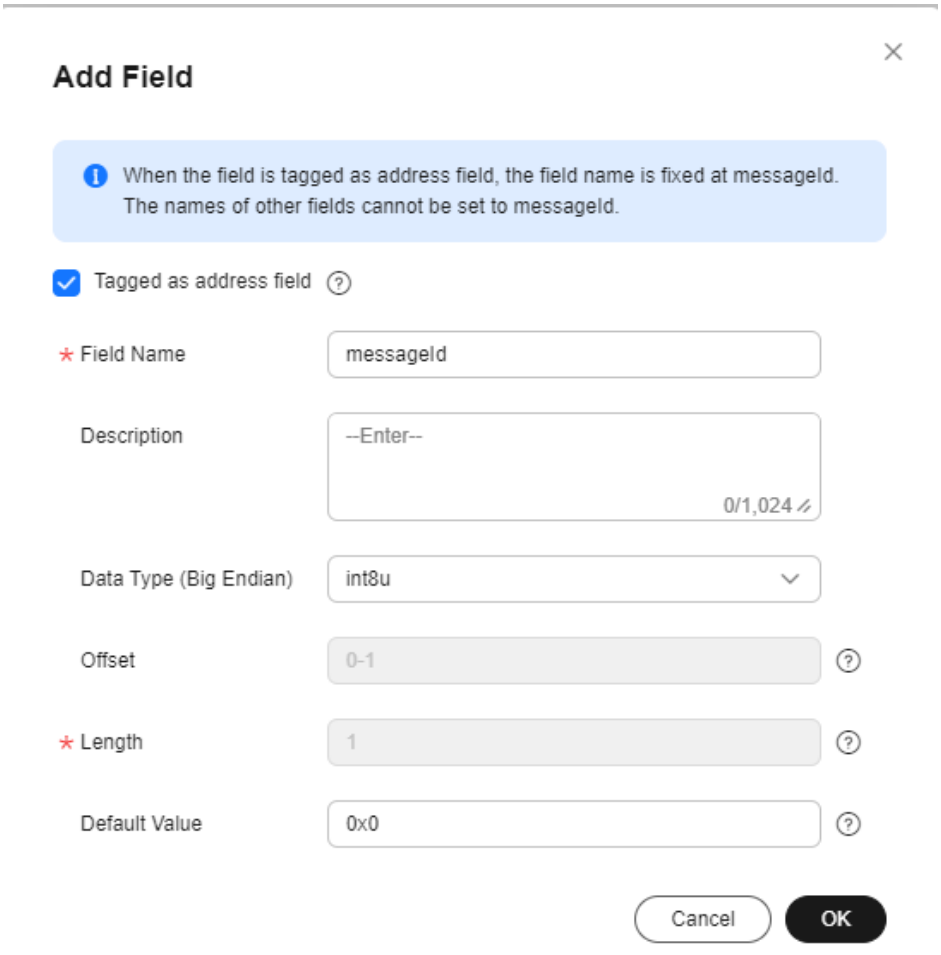


1. Click **Add Field**, select **Tagged as address field**, and add the **messageId** field, which indicates the message type. In this scenario, the message type for reporting the fire severity and temperature is 0x0. When a device reports a message, the first field of each message is **messageId**. For example, if the message reported by a device is 0001013A, the first field **00** indicates that the message is used to report the fire severity and temperature. The subsequent fields **01** and **013A** indicate the fire severity and temperature, respectively. If

there is only one data reporting message and one command delivery message, the **messageId** field does not need to be added.

- **Data Type** is configured based on the number of data reporting message types. The default data type of the **messageId** field is **int8u**.
- The value of **Offset** is automatically filled based on the field location and the number of bytes of the field. **messageId** is the first field of the message. The start position is 0, the byte length is 1, and the end position is 1. Therefore, the value of **Offset** is **0-1**.
- The value of **Length** is automatically filled based on the value of **Data Type**.
- **Default Value** can be changed but must be in hexadecimal format. In addition, the corresponding field in data reporting messages must be the same as the default value.

Figure 3-15 Adding a field - messageId

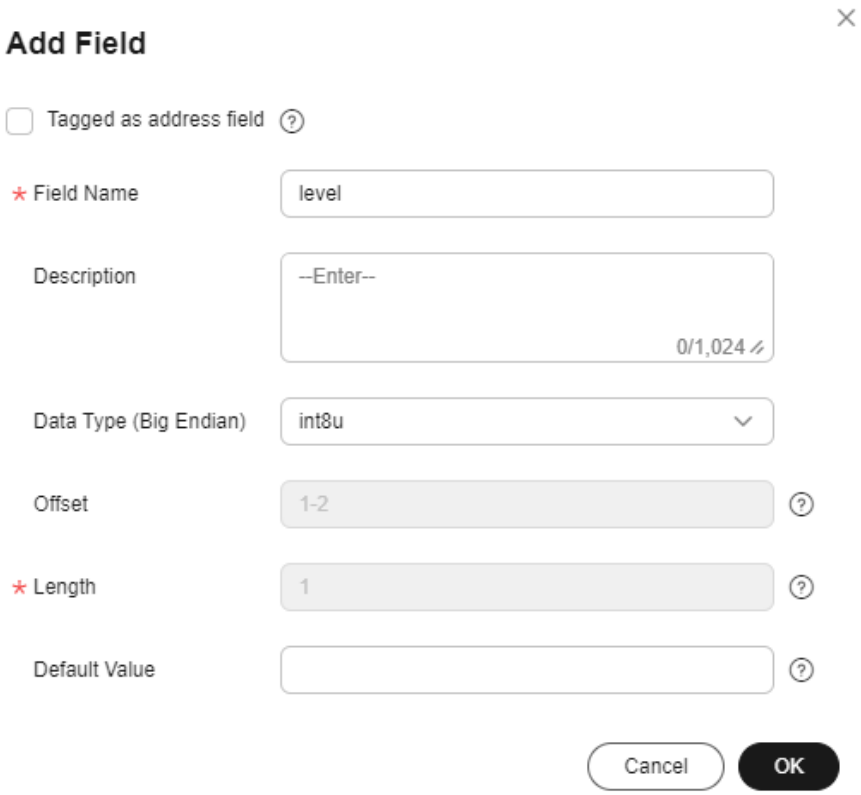


2. Add a **level** field to indicate the fire severity.
   - **Field Name** can contain only letters, digits, underscores (_), and dollar signs ($) and cannot start with a digit.
   - **Data Type** is configured based on the data reported by the device and must match the type defined in the product model. The **level** property

defined in the product model is **int**, and the maximum value is **9**. Therefore, the value of **Data Type** is **int8u**.

- The value of **Offset** is automatically filled based on the field location and the number of bytes of the field. The start position of the **level** field is the end position of the previous field. The end position of the previous field **messageId** is **1**. Therefore, the start position of the **level** field is **1**. The length of the **level** field is 1 byte, and the end position is 2. Therefore, the value of **Offset** is **1-2**.
- The value of **Length** is automatically filled based on **Data Type**.
- **Default Value** can be left blank. If you do not set **Default Value**, the fire level is not fixed and has no default value.

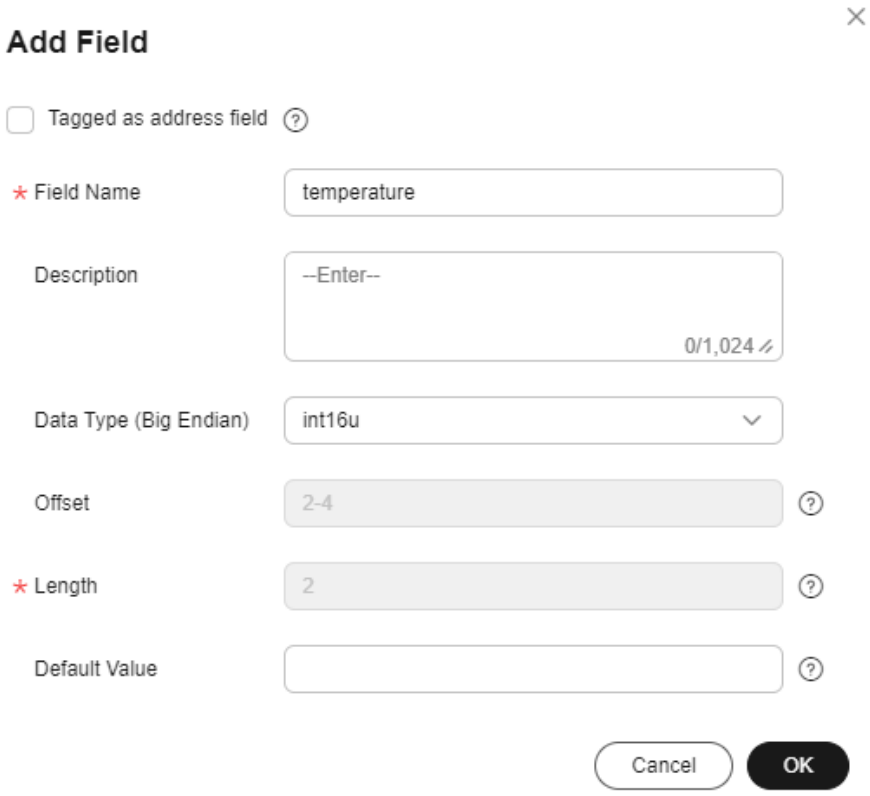
Figure 3-16 Adding a field - level



3. Add the **temperature** field to indicate the temperature at the fire scene.
   - **Data Type**: In the product model, the data type of the **temperature** property is **int** and the maximum value is **1000**. Therefore, the value of **Data Type** is **int16u** in the codec to meet the value range of the **temperature** property.
   - Offset is automatically configured based on the number of characters between the first field and the end field. The start position of the **temperature** field is the end position of the previous field. The end position of the previous field **level** is **2**. Therefore, the start position of the **temperature** field is **2**. The length of the **temperature** field is 2 bytes, and the end position is 4. Therefore, the value of **Offset** is **2-4**.

- The value of **Length** is automatically filled based on **Data Type**.
- If you do not set **Default Value**, the value of the temperature is not fixed and has no default value.

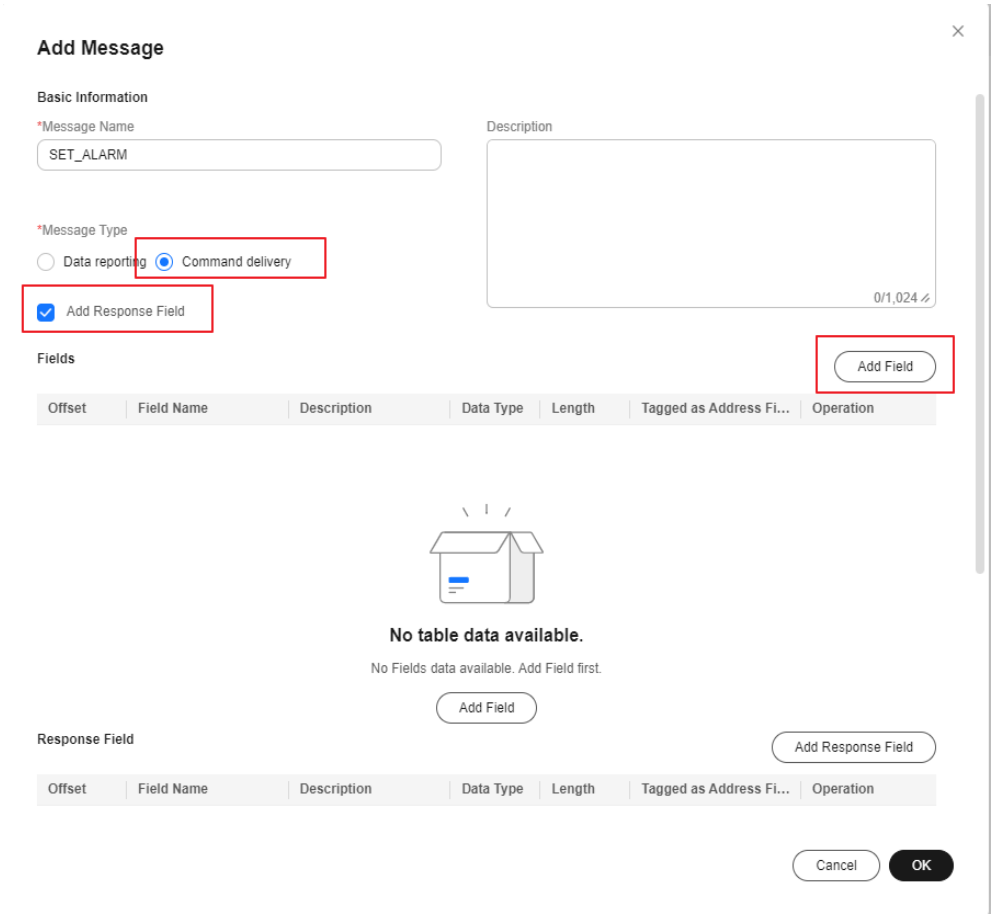
Figure 3-17 Adding a field - temperature



Step 3 Click **Add Message** to add a SET_ALARM message and set the temperature threshold for fire alarms. For example, if the temperature exceeds 60°C, the device reports an alarm. This step is performed to encode the command message in JSON format delivered by the IoT platform into binary data so that the smoke detector can understand the message. The following is a configuration example:

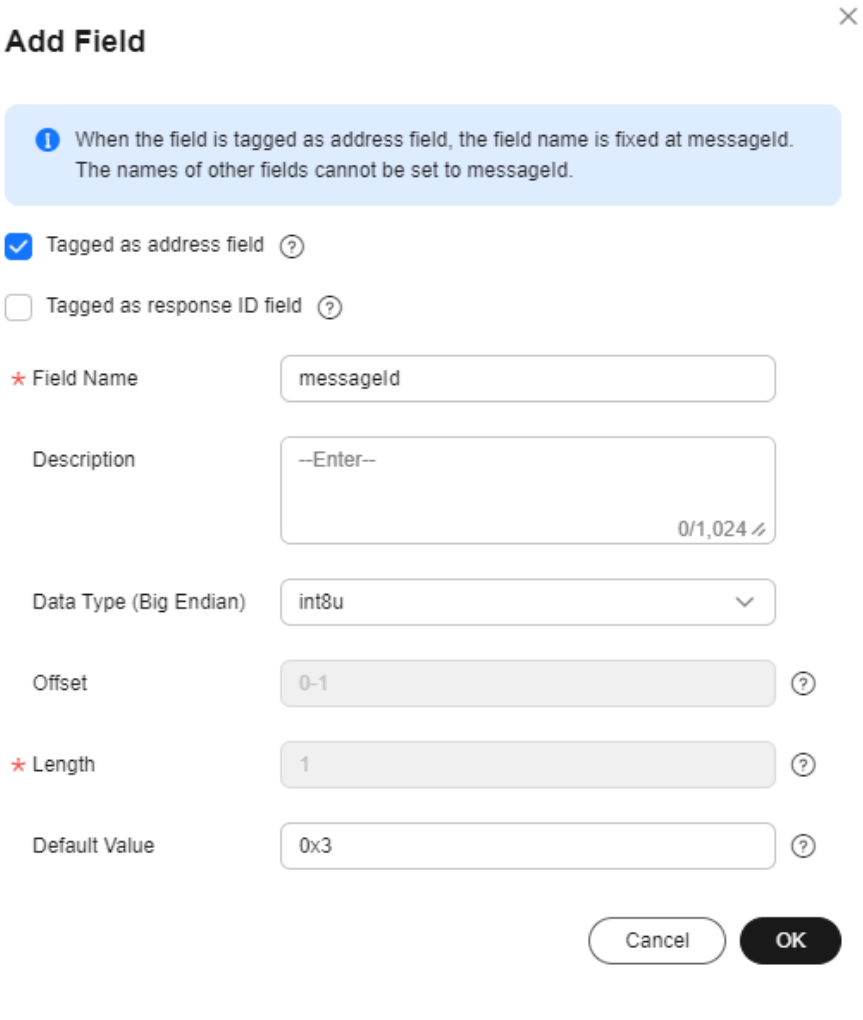
- Message Name: SET_ALARM
- Message Type: Command delivery
- Add Response Field: selected. After a response field is added, the device reports the command execution result after receiving the command. You can determine whether to add response fields as required.

Figure 3-18 Adding a message - SET_ALARM



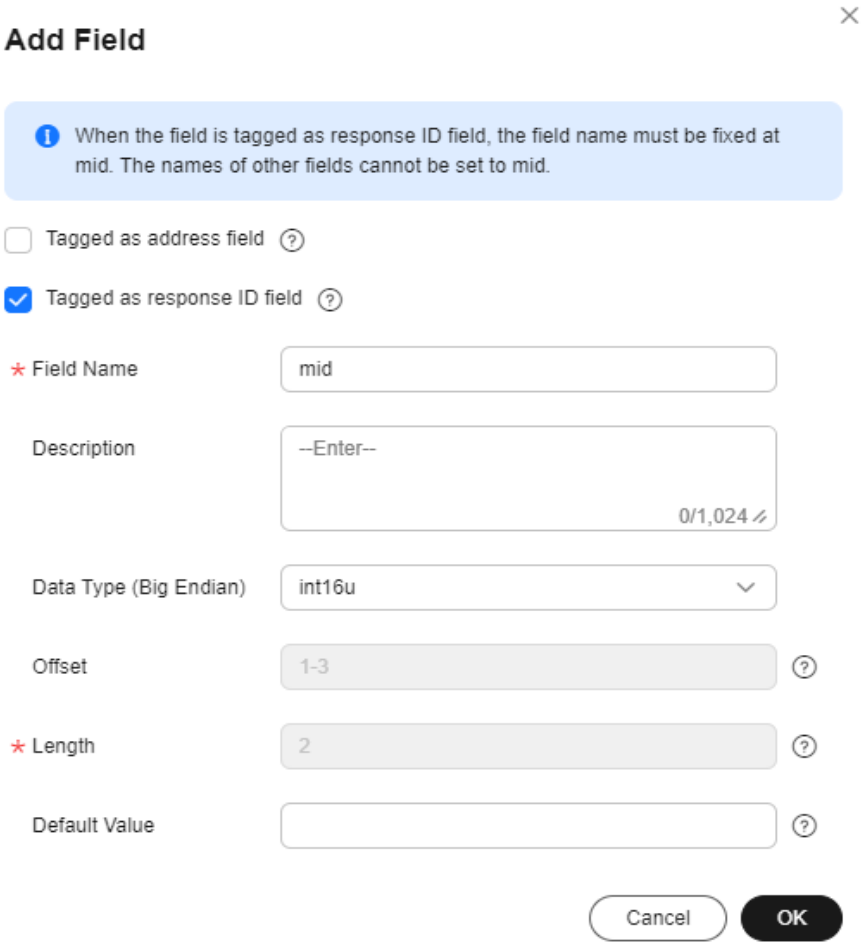
a. Click **Add Field** to add the **messageId** field, which indicates the message type. For example, set the message type of the fire alarm threshold to **0x3**. For details about the message ID, data type, length, default value, and offset, see **1**.

Figure 3-19 Adding a command field - messageId (0x3)



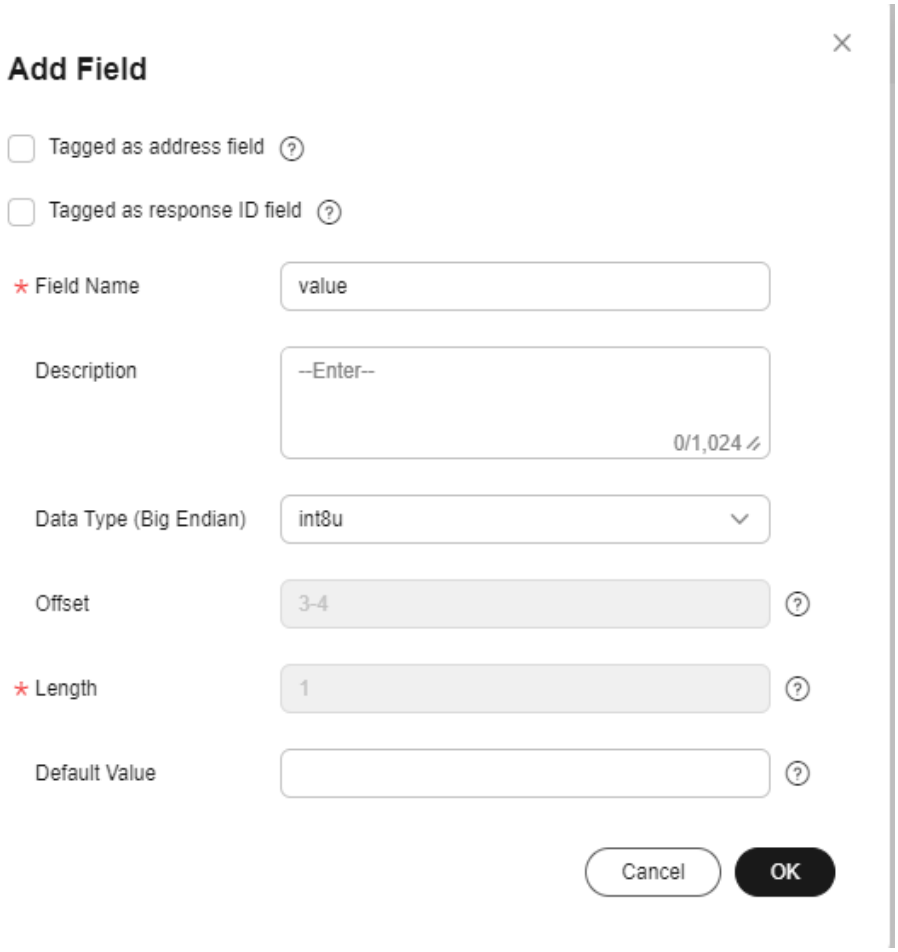
b. Add the **mid** field. This field is generated and delivered by the platform and is used to associate the delivered command with the command delivery response. The data type of the **mid** field is **int16u** by default. For details about the length, default value, and offset, see 2.

Figure 3-20 Adding a command field - mid



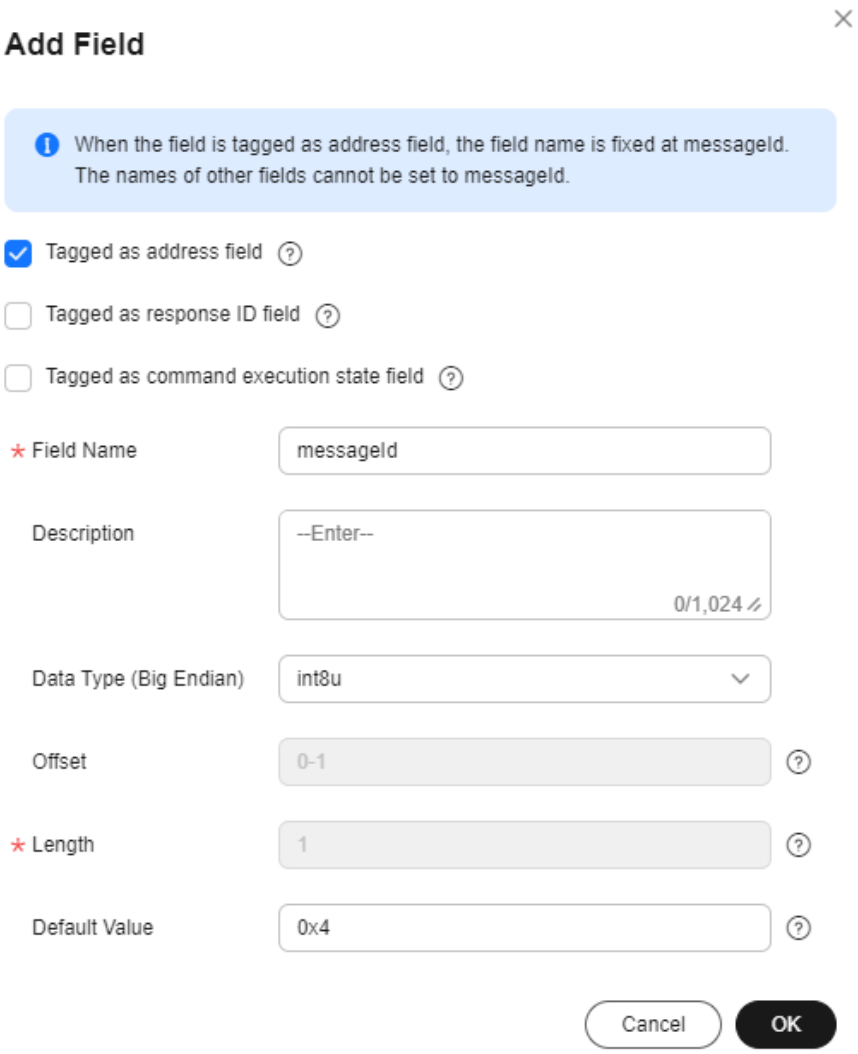
c. Add the **value** field to indicate the parameter value of the delivered command. For example, deliver the temperature threshold for a fire alarm. For details about the data type, length, default value, and offset, see 2.

Figure 3-21 Adding a command field - value



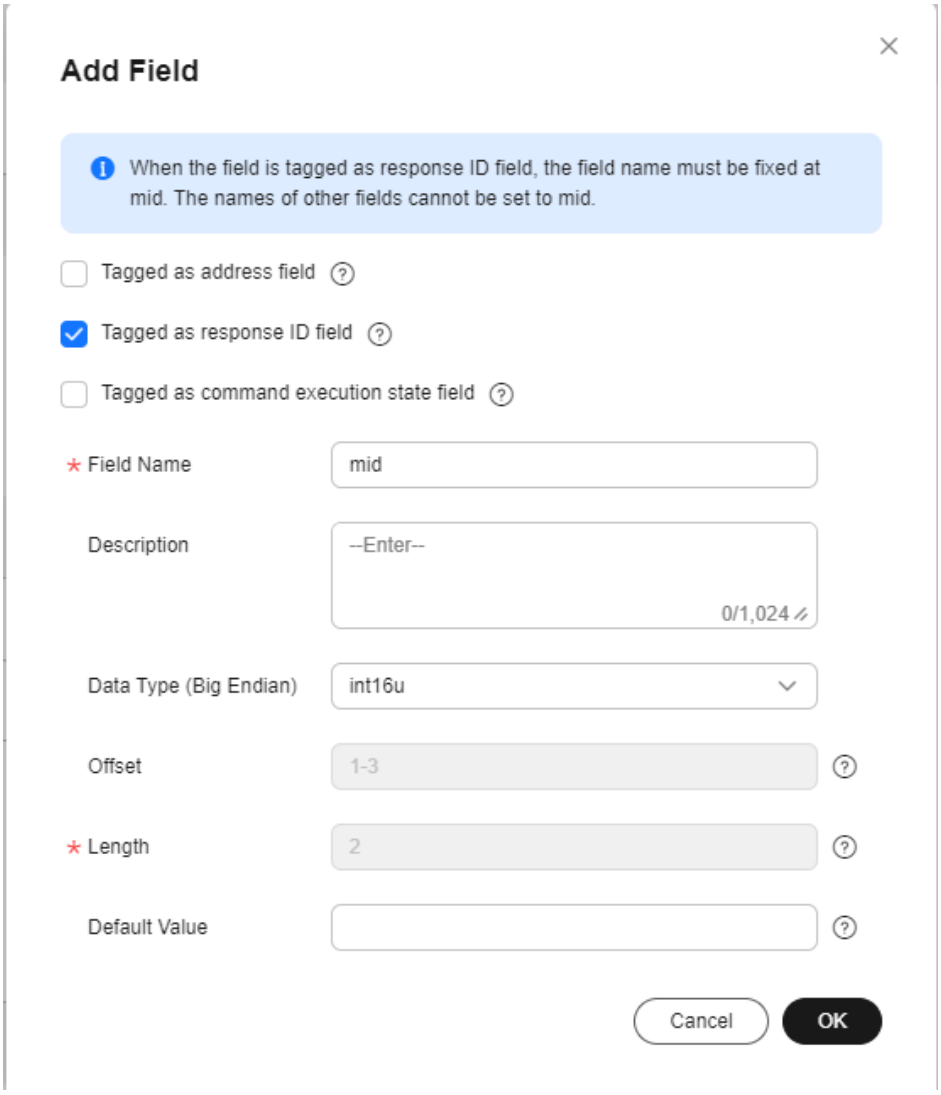
d. Click **Add Response Field** to add the **messageId** field, which indicates the message type. The command delivery response is an upstream message, which is differentiated from the data reporting message by the **messageId** field. The message type for reporting the temperature threshold of the fire alarm is **0x4**. For details about the message ID, data type, length, default value, and offset, see **1**.

Figure 3-22 Adding a response field - messageId (0x4)



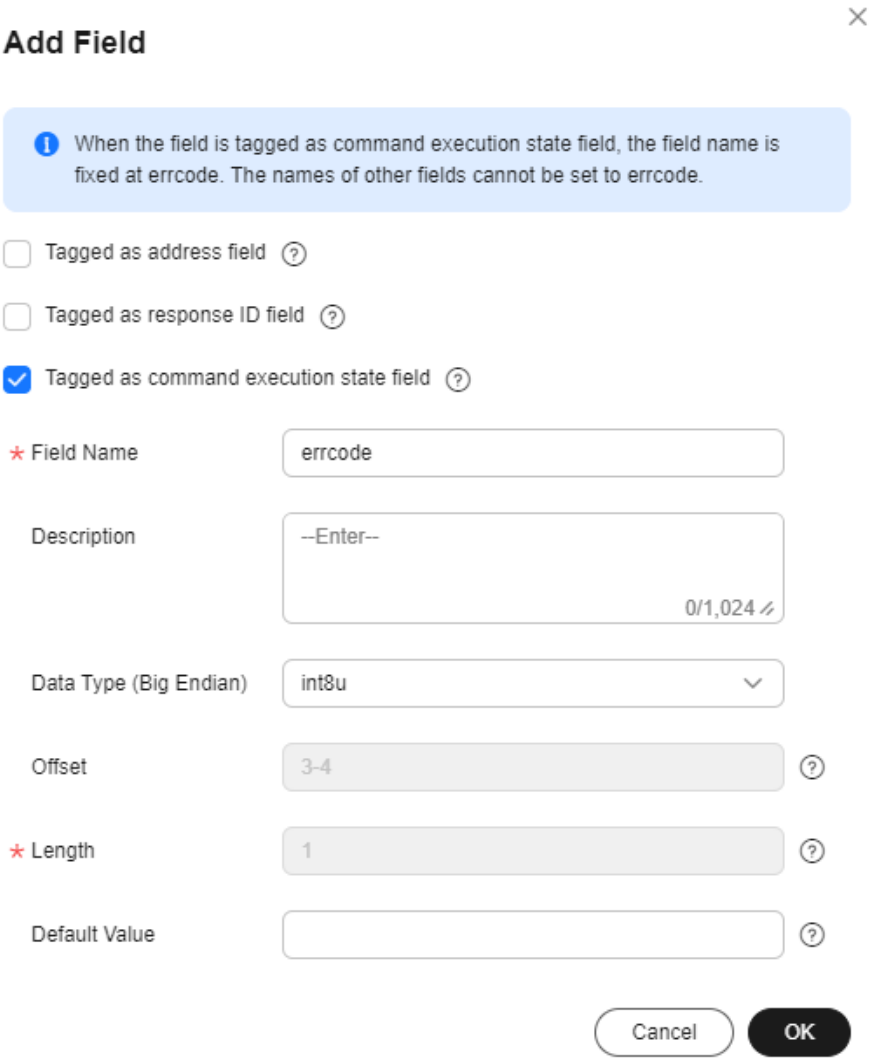
e. Add the **mid** field. This field must be the same as that in the command delivered by the IoT platform. It is used to associate the delivered command with the command execution result. The data type of the **mid** field is **int16u** by default. For details about the length, default value, and offset, see 2.

Figure 3-23 Adding a response field - mid



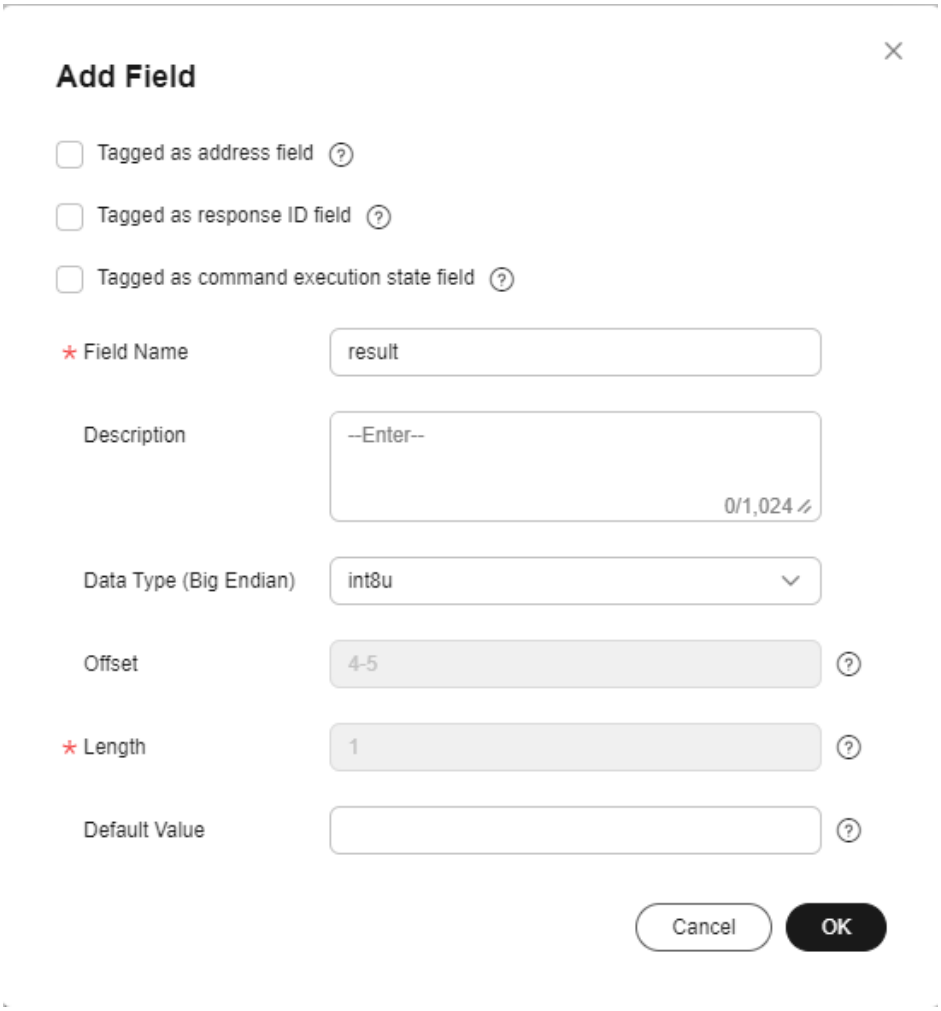
f. Add the **errcode** field to indicate the command execution status. **00** indicates success and **01** indicates failure. If this field is not carried in the response, the command is executed successfully by default. The data type of the **errcode** field is **int8u** by default. For details about the length, default value, and offset, see **2**.

Figure 3-24 Adding a response field - errcode



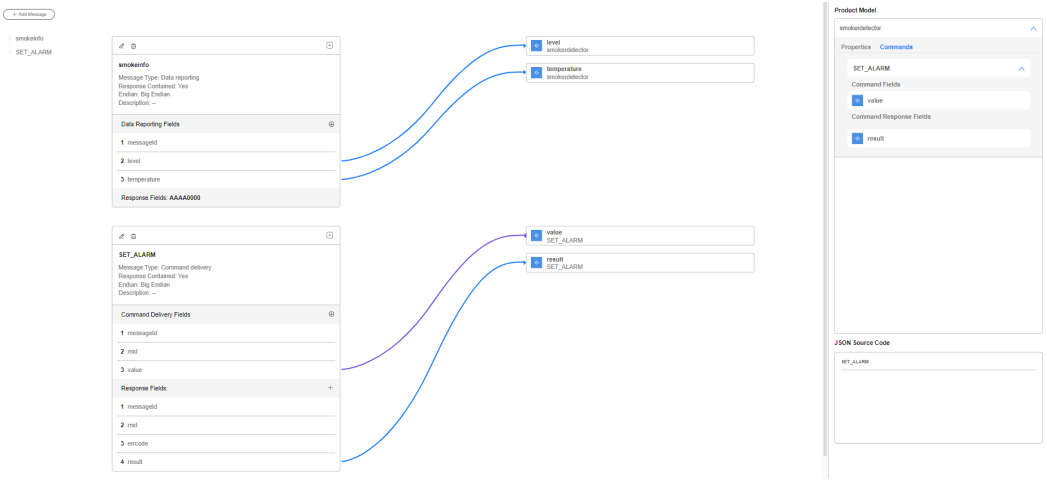
g. Add the **result** field to indicate the command execution result. For example, the device returns the current alarm threshold to the platform.

Figure 3-25 Adding a response field - result



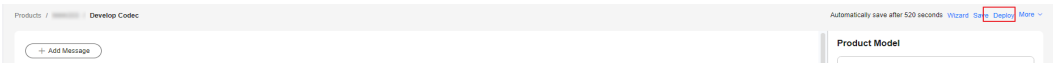
Step 4 Drag the property fields and command fields in **Device Model** on the right to set up a mapping between the fields in the data reporting message and those in the command delivery message.

Figure 3-26 Developing a codec online - smokerdetector



Step 5 Click **Save** and then **Deploy** to deploy the codec on the platform.

Figure 3-27 Deploying a codec



----End

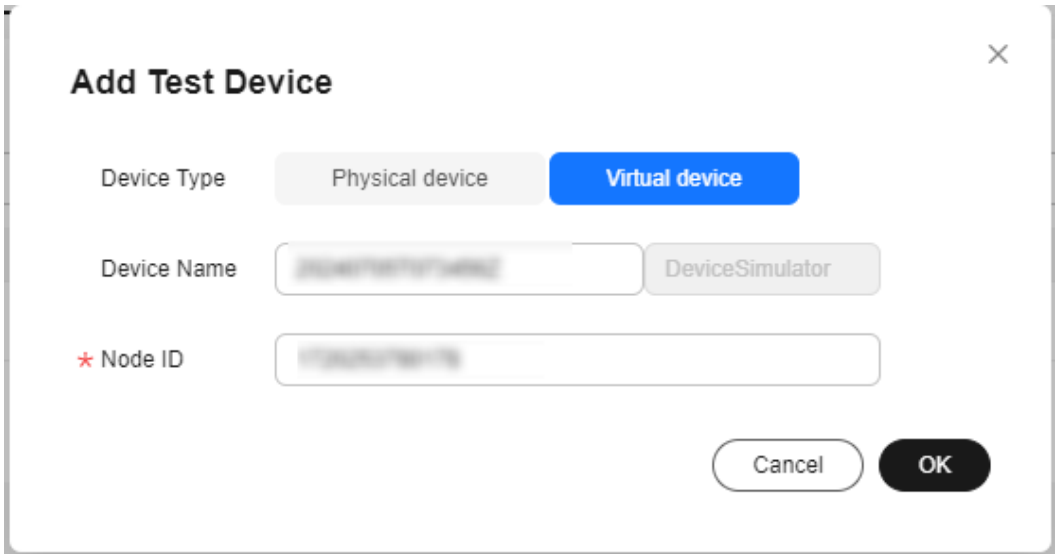
Testing the Codec

Step 1 On the product details page of the smoke detector, click the **Online Debugging** tab and click **Add Test Device**.

Step 2 You can use a real device or virtual device for debugging based on your service scenario. For details, see **Online Debugging**. The following uses a virtual device as an example to describe how to debug a codec.

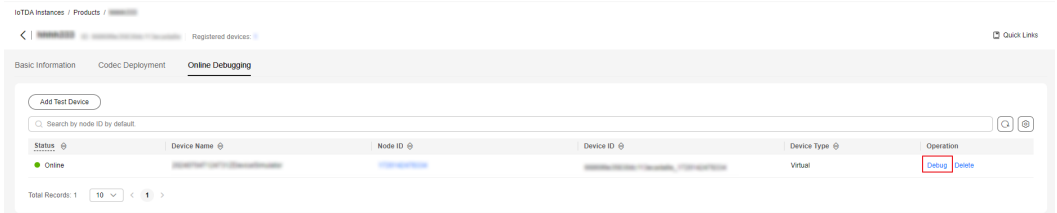
In the **Add Test Device** dialog box, select **Virtual device** for **Device Type** and click **OK**. The virtual device name contains **DeviceSimulator**. Only one virtual device can be created for each product.

Figure 3-28 Online debugging - Creating a virtual device



Step 3 Click **Debug** to access the debugging page.

Figure 3-29 Entering debugging

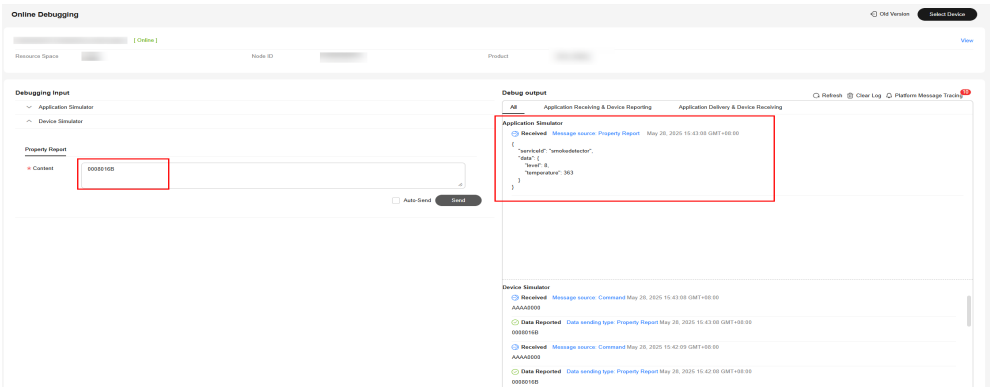


Step 4 Use the device simulator to report data. For example, a hexadecimal code stream (0008016B) is reported. **00** indicates the **messageId** field. **08** indicates the fire level, and its length is one byte. **016B** indicates the temperature, and its length is two bytes.

View the data reporting result ({level=8, temperature=363}) in **Application Simulator**. 8 is the decimal number converted from the hexadecimal number 08 and 363 from the hexadecimal number 016B.

In the **Device Simulator** area, the response data AAAA0000 delivered by the IoT platform is displayed.

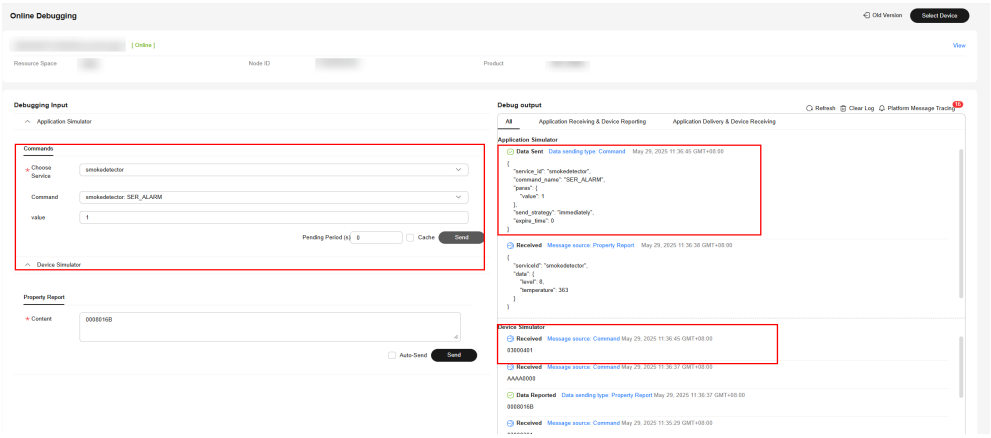
Figure 3-30 Online debugging - Simulating data reporting (smokerdetector)



Step 5 Use the application simulator to deliver a command and set **value** to **1**. The command {"serviceId": "Smokeinfo", "method": "SET_ALARM", "paras": "{\"value\":1}"} is delivered.

View the command receiving result in **Device Simulator**, which is **03000101**. **03** indicate the **messageId** field, **0001** indicates the **mid** field, and **01** is the hexadecimal value converted from the decimal value **1**.

Figure 3-31 Online debugging - Simulating command delivery (smokerdetector)



NOTE

During online debugging of a CoAP virtual device, if the device simulator does not receive the delivered command, use the device simulator to report the property, and deliver the command again.

----End

Summary

- If the codec needs to parse the command execution result, the **mid** field must be defined in the command and the command response.
- The length of the **mid** field in a command is two bytes. For each device, **mid** increases from 1 to 65535, and the corresponding code stream ranges from 0001 to FFFF.
- After a command is executed, the **mid** field in the reported command execution result must be the same as that in the delivered command. In this way, the IoT platform can update the command status.

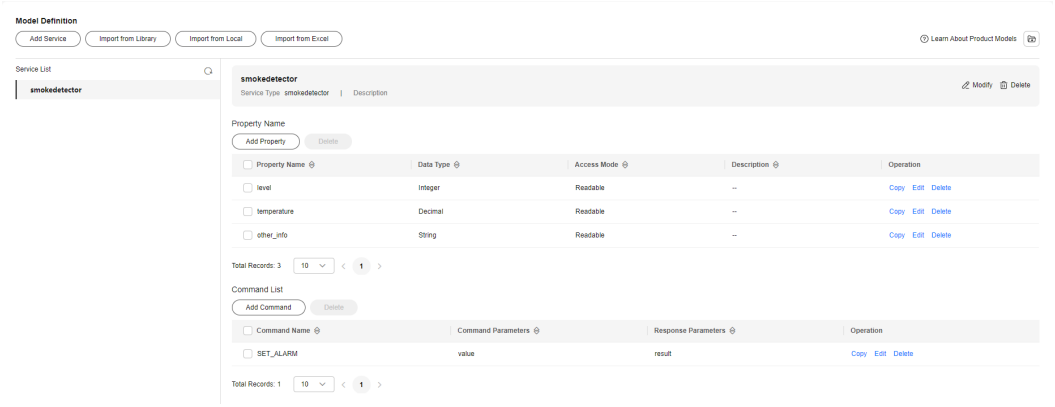
Codec for Strings and Variable-Length Strings

If the smoke detector needs to report the description information in strings or variable-length strings, perform the following steps to create messages:

Product Model

Create a smoke sensor product and define the product model on the product details page.

Figure 3-32 Model definition - smokedetector carrying other_info



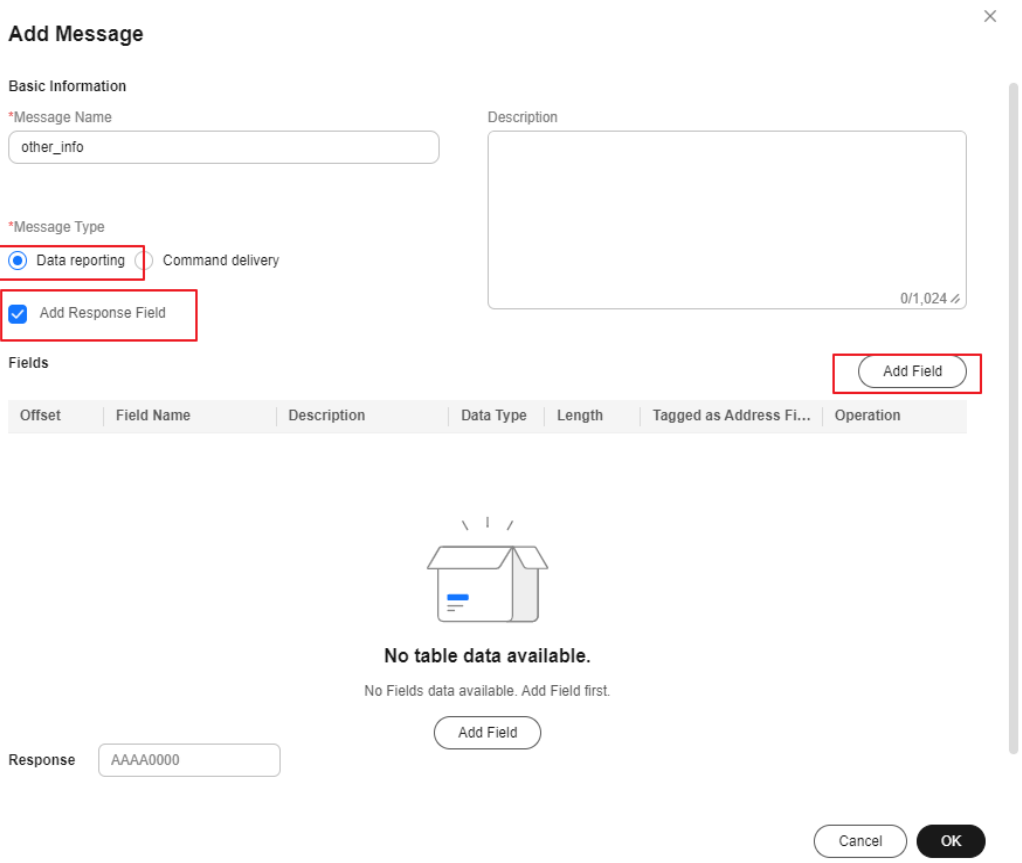
Developing a Codec

Step 1 On the smoke detector details page, click the **Codec Development** tab and click **Develop Codec**.

Step 2 Click **Add Message** to add the **other_info** message and report the description of the string type. This step is performed to decode the binary code stream message of the string uploaded by the device to the JSON format so that the platform can understand the message. The following is a configuration example:

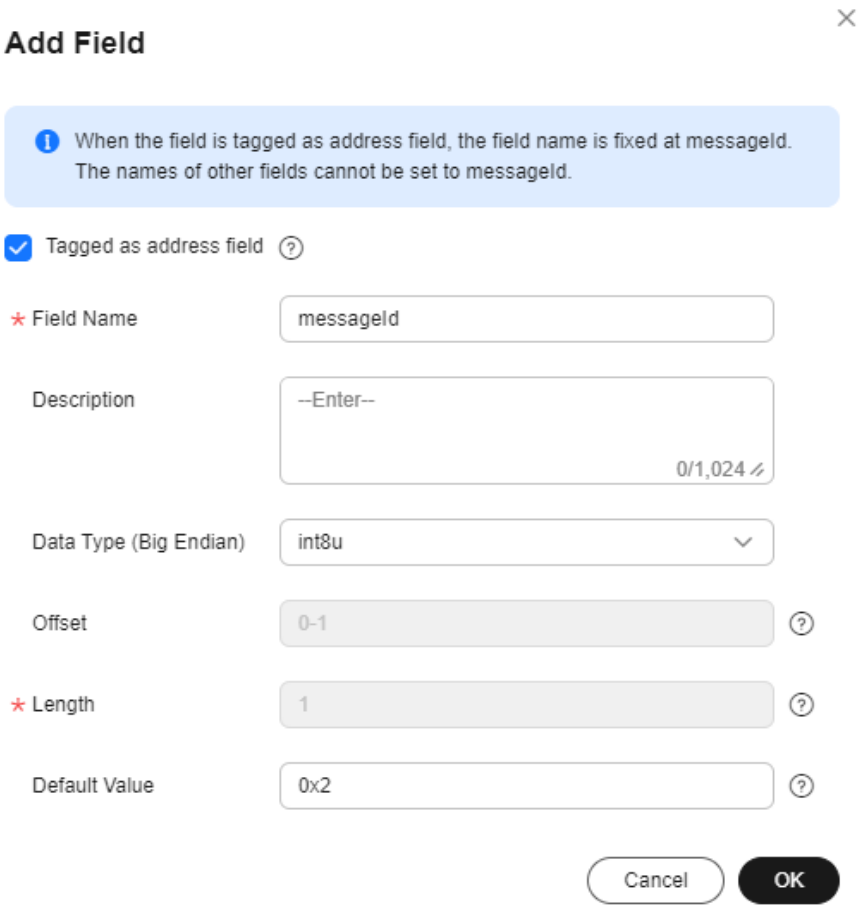
- **Message Name**: **other_info**
- **Message Type**: **Data reporting**
- **Add Response Field**: selected. After response fields are added, the platform delivers the response data set by the application to the device after receiving the data reported by the device.
- **Response**: AAAA0000 (default)

Figure 3-33 Adding a message - other_info



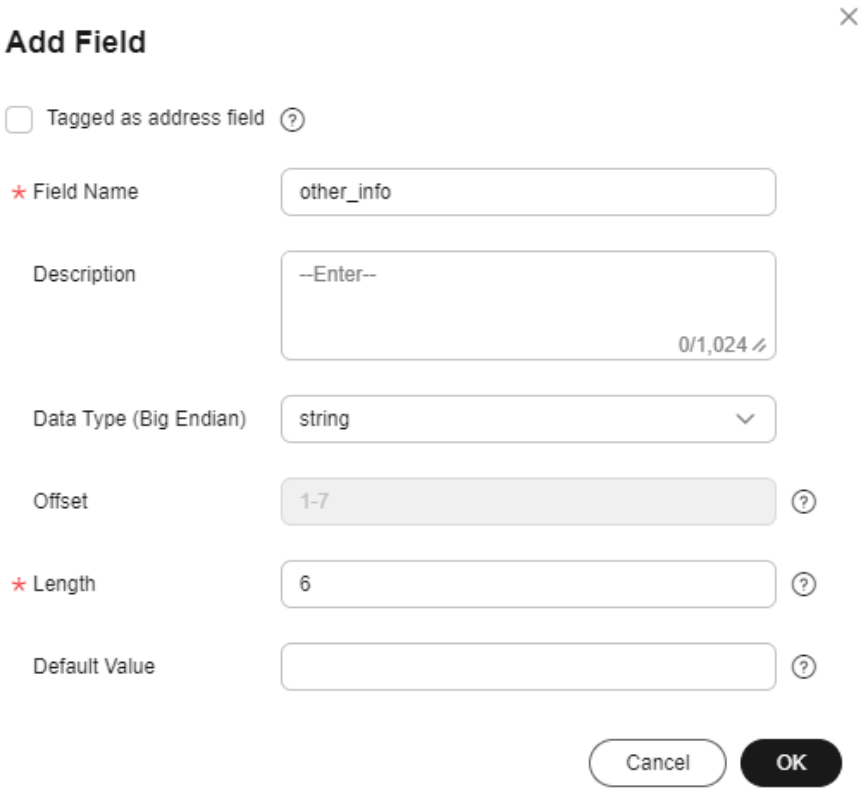
1. Click **Add Field** to add the **messageId** field, which indicates the message type. In this scenario, the value **0x0** is used to identify the message that reports the fire severity and temperature, **0x1** is used to identify the message that reports only the temperature, and **0x2** is used to identify the message that reports the description (of the string type). For details about the message ID, data type, length, default value, and offset, see 1.

Figure 3-34 Adding a field - messageId (0x2)



2. Add the **other_info** field to indicate the description of the string type. In this scenario, set **Data Type** to **string** and **Length** to **6**. For details about the field name, default value, and offset, see **2**.

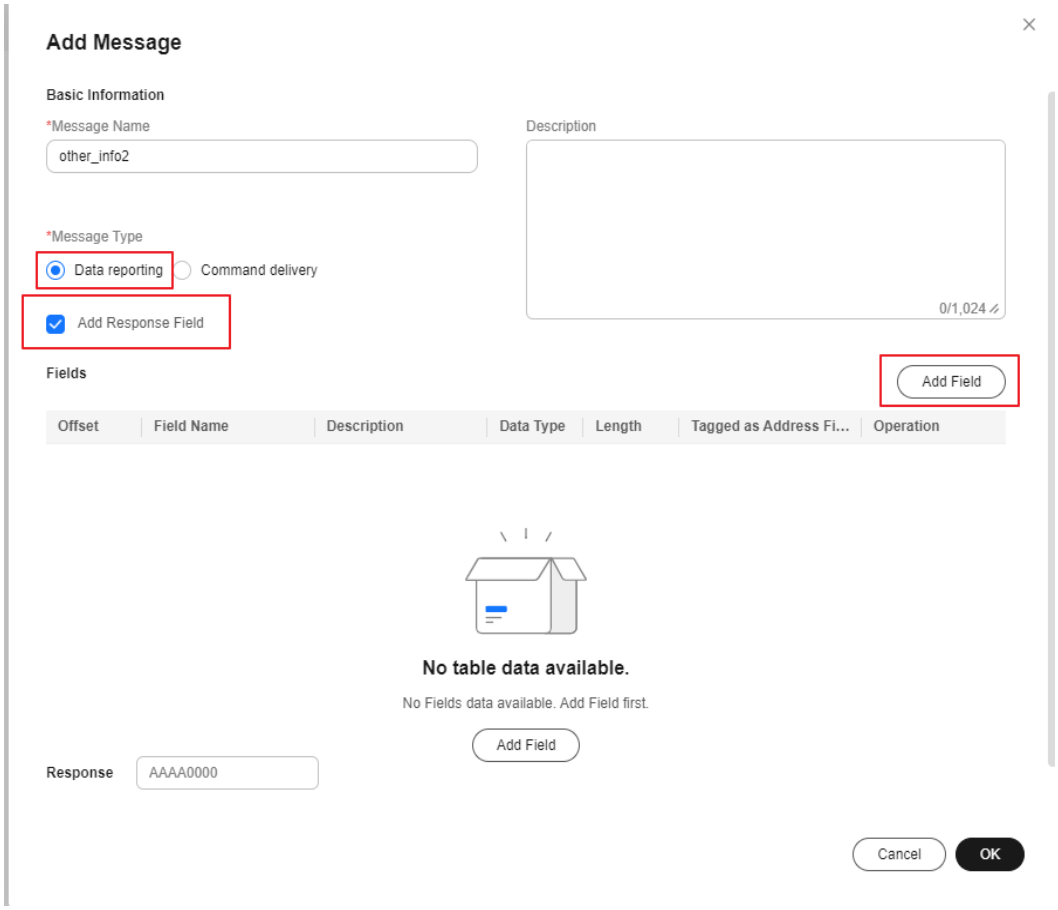
Figure 3-35 Adding a field - other_info



Step 3 Click **Add Message**, add the **other_info2** message name, and configure the data reporting message to report the description of the variable-length string type. This step is performed to decode the binary code stream message of variable-length strings uploaded by the device to the JSON format so that the platform can understand the message. The following is a configuration example:

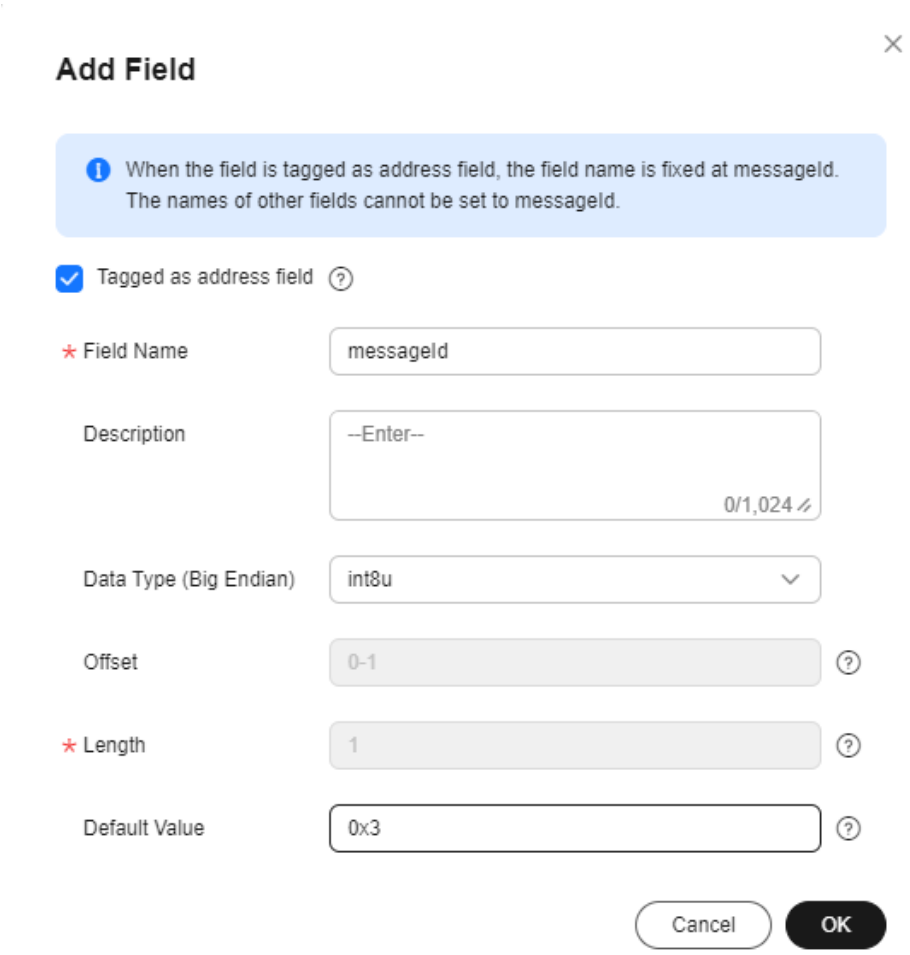
- **Message Name**: other_info2
- **Message Type**: Data reporting
- **Add Response Field**: selected. After response fields are added, the platform delivers the response data set by the application to the device after receiving the data reported by the device.
- **Response**: AAAA0000 (default)

Figure 3-36 Adding a message - other_info2



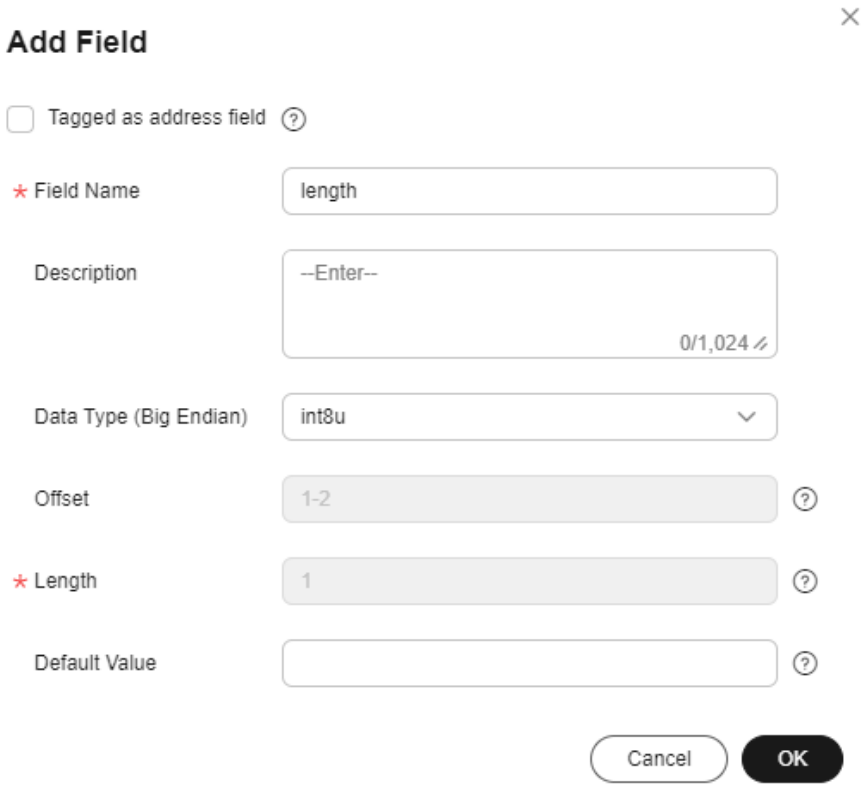
1. Add the **messageId** field to indicate the message type. In this scenario, the value **0x0** is used to identify the message that reports the fire severity and temperature, **0x1** is used to identify the message that reports only the temperature, and **0x3** is used to identify the message that reports the description (of the variable-length string type). For details about the message ID, data type, length, default value, and offset, see **1**.

Figure 3-37 Adding a field - messageId (0x3)



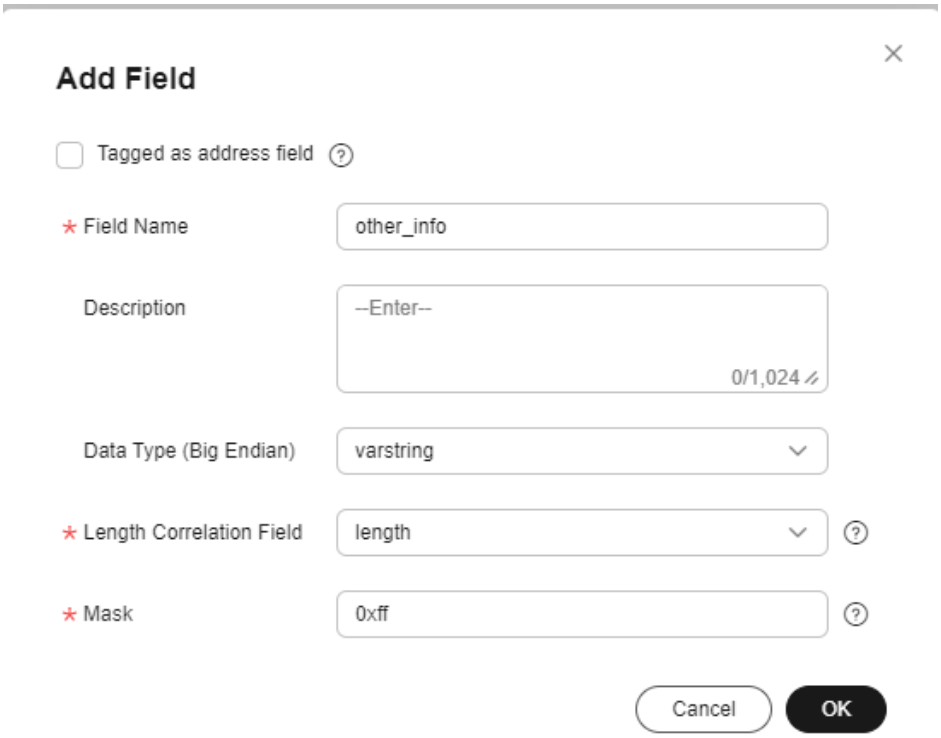
2. Add the **length** field to indicate the length of a variable-length string. **Data Type** is configured based on the length of the variable-length string. If the string contains 255 or fewer characters in this scenario, set this parameter to **int8u**. For details about the length, default value, and offset, see 2.

Figure 3-38 Adding a field - length



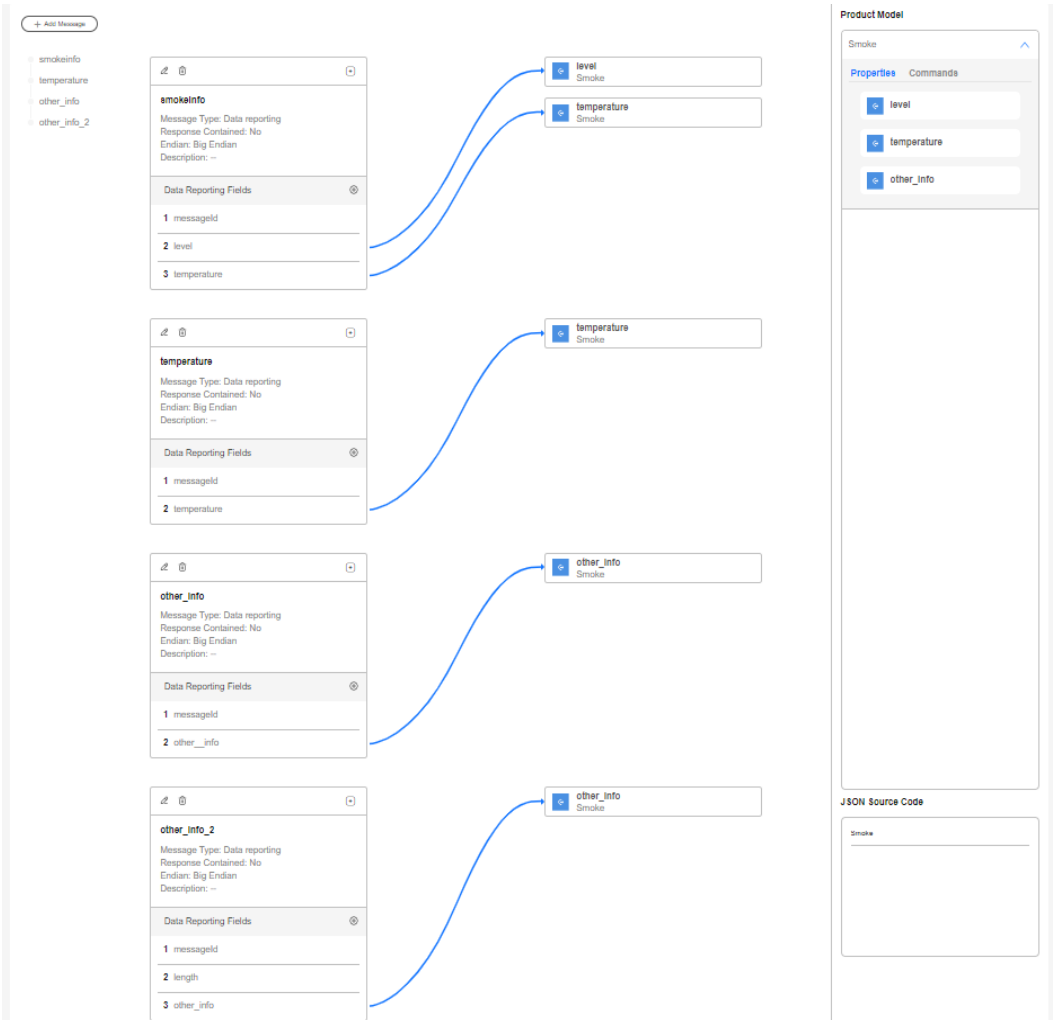
3. Add the **other_info** field and set **Data Type** to **varstring**, which indicates the description of the variable-length string type. Set **Length Correlation Field** to **length**, indicating that the length of the current variable-length string is determined by the reported value of length. The default mask is **0xff**, which is used to calculate the actual length of the field. For example, if the value of **Length Correlation Field** is 5, the binary value is **00000101**. If the mask is **0xff**, the binary value is **11111111**. The result of the AND operation on these two values is **00000101**, that is, **5** in decimal format. Therefore, the length of this field that takes effect is 5 bytes. For example, if the reported data is **03051234567890**, its message ID is **03**, its length is 5 bytes, and the code stream corresponding to **other_info** is **1234567890**.

Figure 3-39 Adding a field - other_info as varstring



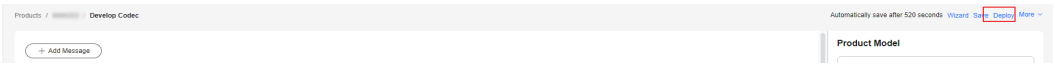
Step 4 Drag the property fields in **Device Model** on the right to set up a mapping between the corresponding fields in the data reporting messages.

Figure 3-40 Developing a codec - Data reporting field mapping



Step 5 Click **Save** and then **Deploy** to deploy the codec on the platform.

Figure 3-41 Deploying a codec



----End

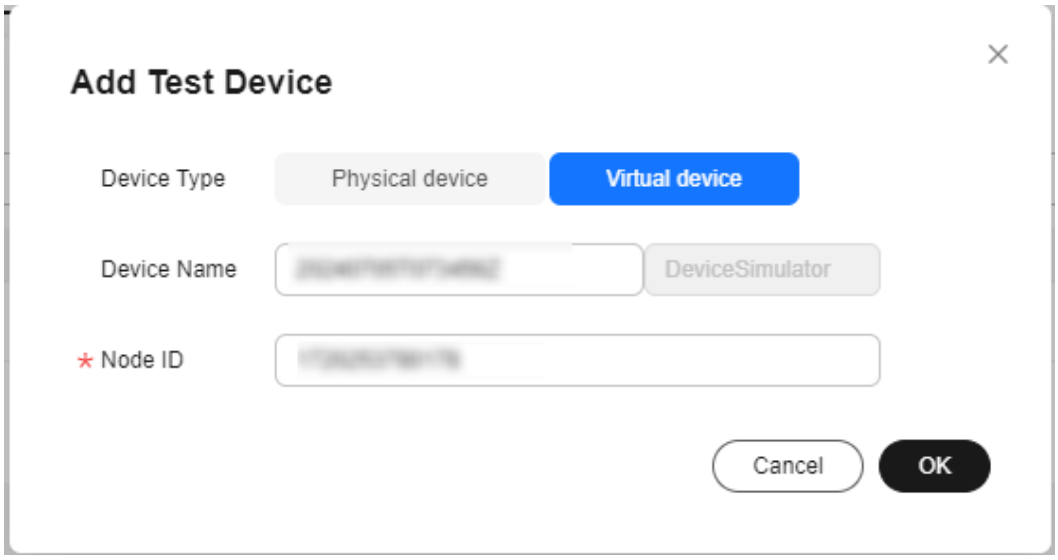
Testing the Codec

Step 1 On the product details page of the smoke detector, click the **Online Debugging** tab and click **Add Test Device**.

Step 2 You can use a real device or virtual device for debugging based on your service scenario. For details, see **Online Debugging**. The following uses a virtual device as an example to describe how to debug a codec.

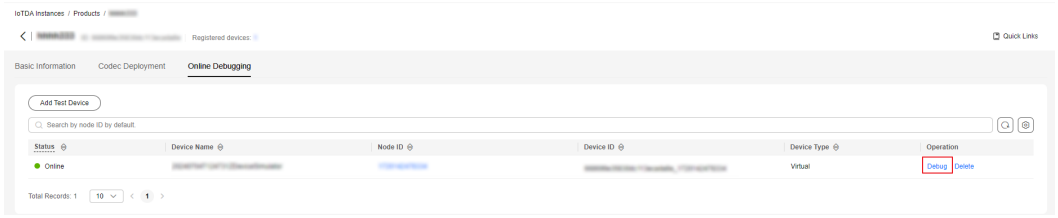
In the **Add Test Device** dialog box, select **Virtual device** for **Device Type** and click **OK**. The virtual device name contains **DeviceSimulator**. Only one virtual device can be created for each product.

Figure 3-42 Online debugging - Creating a virtual device



Step 3 Click **Debug** to access the debugging page.

Figure 3-43 Entering debugging

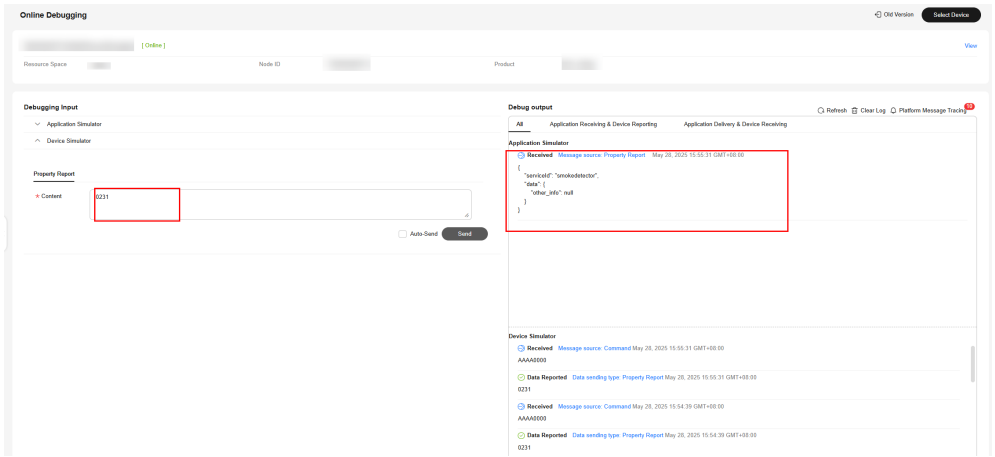


Step 4 Use the device simulator to report the description of the string type.

In the hexadecimal code stream example (0231), **02** indicates the **messageId** field and specifies that this message reports the description of the string type. **31** indicates the description and its length is one byte.

View the data reporting result ({other_info=null}) in **Application Simulator**. The length of the description is less than six bytes. Therefore, the codec cannot parse the description.

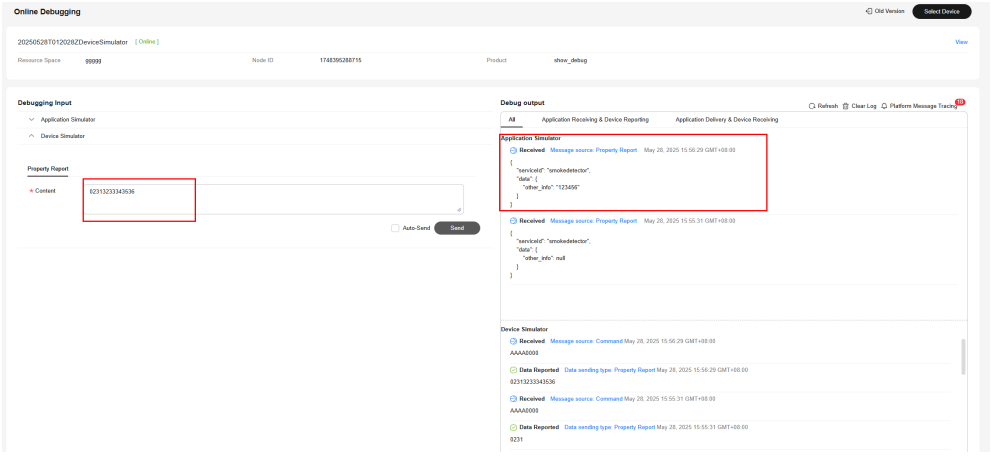
Figure 3-44 Simulating data reporting - other_info too short



In the hexadecimal code stream example (02313233343536), **02** indicates the **messageId** field and specifies that this message reports the description of the string type. **313233343536** indicates the description and its length is six bytes.

View the data reporting result ({other_info=123456}) in **Application Simulator**. The length of the description is six bytes. The description is parsed successfully by the codec.

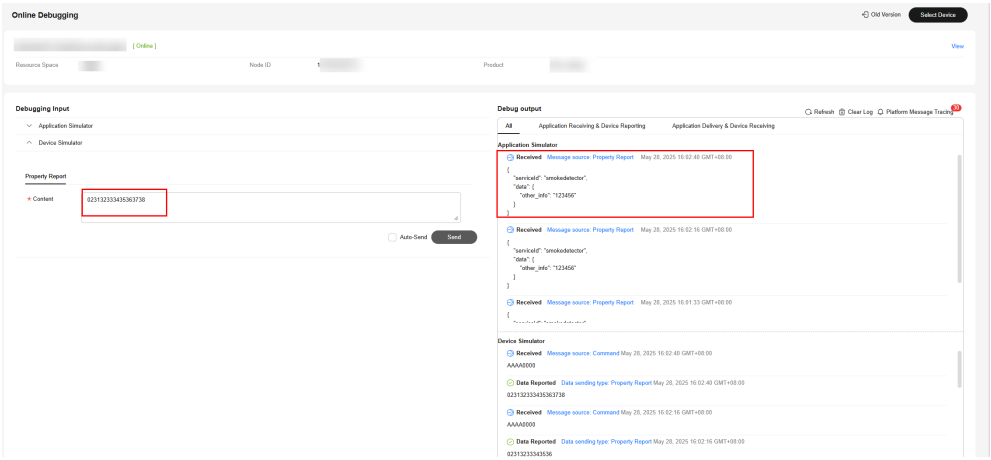
Figure 3-45 Simulating data reporting - other_info length proper



In the hexadecimal code stream example (023132333435363738), **02** indicates the **messageId** field and specifies that this message reports the description of the string type. **3132333435363738** indicates the description and its length is eight bytes.

View the data reporting result ({other_info=123456}) in **Application Simulator**. The length of the description exceeds six bytes. Therefore, the first six bytes are intercepted and parsed by the codec.

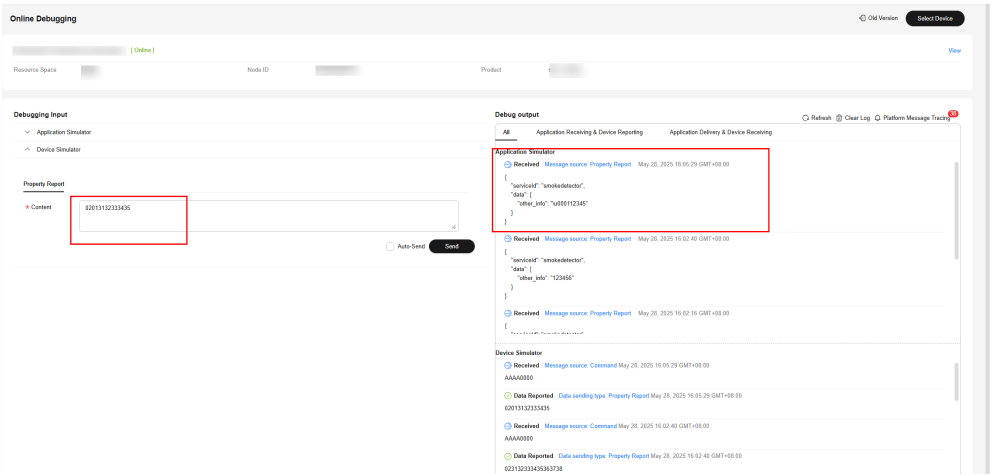
Figure 3-46 Simulating data reporting - other_info too long



In the hexadecimal code stream example (02013132333435), **02** indicates the **messageId** field and specifies that this message reports the description of the string type. **013132333435** indicates the description and its length is six bytes.

View the data reporting result ({other_info=\u000112345}) in **Application Simulator**. In the ASCII code table, **01** indicates **start of headline** which cannot be represented by specific characters. Therefore, 01 is parsed to \u0001.

Figure 3-47 Simulating data reporting - other_info as ASCII code

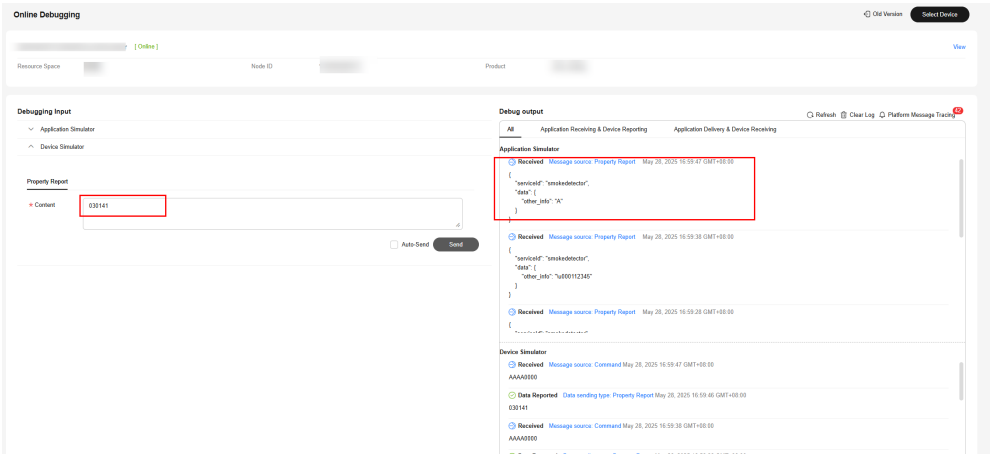


Step 5 Use the device simulator to report the description of the variable-length string type.

In the hexadecimal code stream example (030141), **03** indicates the **messageId** field and specifies that this message reports the description of the variable-length string type. **01** indicates the length of the description. **41** indicates the description content and its length is one byte.

View the data reporting result ({other_info=A}) in **Application Simulator**. A corresponds to 41 in the ASCII code table.

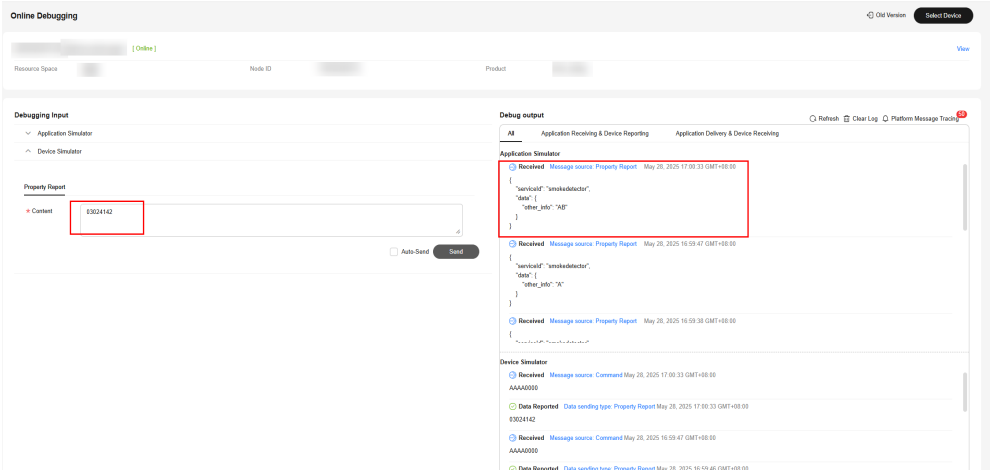
Figure 3-48 Simulating data reporting - other_info as variable-length character string 1



In the hexadecimal code stream example (03024142), **03** indicates the **messageId** field and specifies that this message reports the description of the variable-length string type. **02** indicates the length of the description. **4142** indicates the description content and its length is two bytes.

View the data reporting result ({other_info=AB}) in **Application Simulator**. A corresponds to 41 and B corresponds to 42 in the ASCII code table.

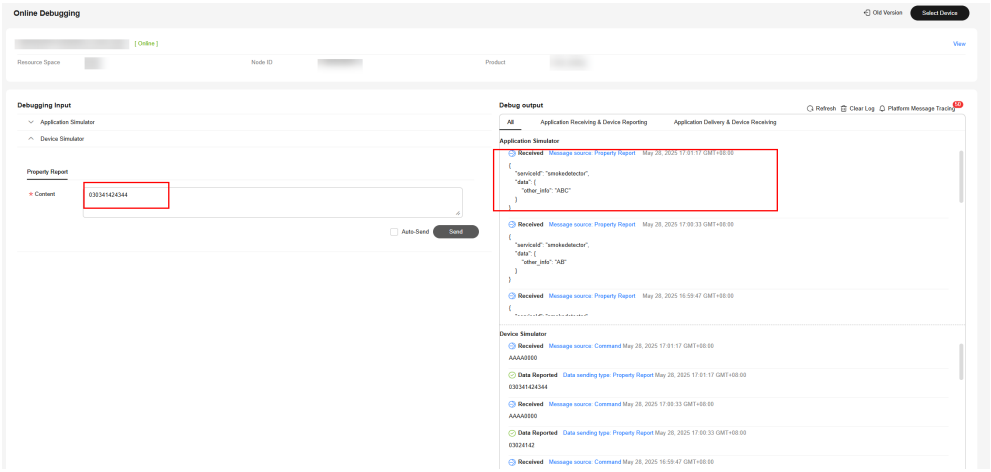
Figure 3-49 Simulating data reporting - other_info as variable-length character string 2



In the hexadecimal code stream example (030341424344), **03** indicates the **messageId** field and specifies that this message reports the description of the variable-length string type. The second **03** indicates the length of the description. **41424344** indicates the description content and its length is four bytes.

View the data reporting result ({other_info=ABC}) in **Application Simulator**. The length of the description exceeds three bytes. Therefore, the first three bytes are intercepted and parsed. In the ASCII code table, A corresponds to 41, B to 42, and C to 43.

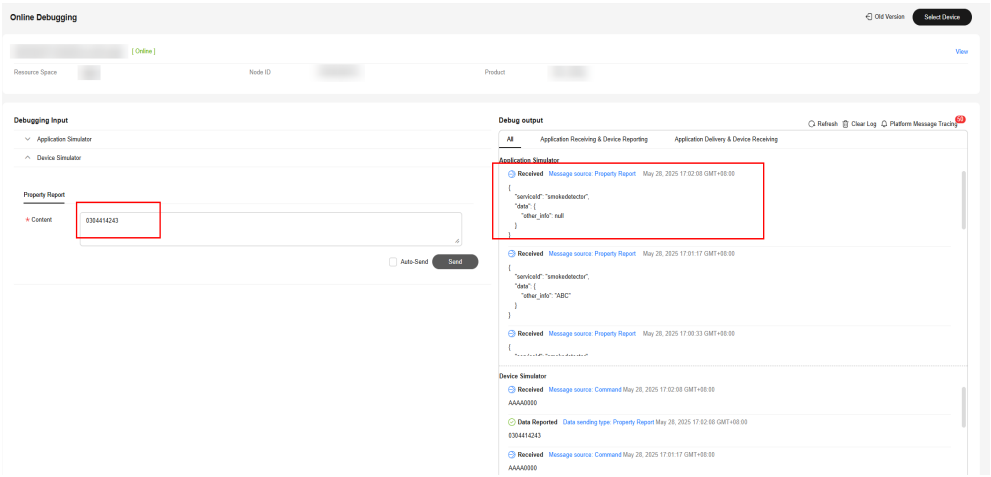
Figure 3-50 Simulating data reporting - other_info as variable-length character string 3



In the hexadecimal code stream example (0304414243), **03** indicates the **messageId** field and specifies that this message reports the description of the variable-length string type. **04** indicates the string length (four bytes) and its length is one byte. **414243** indicates the description and its length is four bytes.

View the data reporting result ({other_info=null}) in **Application Simulator**. The length of the description is less than four bytes. The codec fails to parse the description.

Figure 3-51 Simulating data reporting - other_info as variable-length character string 4



----End

Summary

- When data is a string or a variable-length string, the codec processes the data based on the ASCII code. When data is reported, the hexadecimal code stream is decoded to a string. For example, 21 is parsed to an exclamation mark (!), 31 to 1, and 41 to A. When a command is delivered, the string is encoded into a hexadecimal code stream. For example, an exclamation mark (!) is encoded into 21, 1 into 31, and A into 41.
- When the data type of a field is **varstring** (variable-length string type), the field must be associated with the **length** field. The data type of the **length** field must be **int**.
- For variable-length strings, the codecs for command delivery and data reporting are developed in the same way.
- Codecs developed online encode and decode strings and variable-length strings using the ASCII hexadecimal standard table. During decoding (data reporting), if the parsing results cannot be represented by specific characters such as start of headline, start of text, and end of text, the \u+2 byte code stream values are used to indicate the results. For example, 01 is parsed to \u0001 and 02 to \u0002. If the parsing results can be represented by specific characters, specific characters are used.

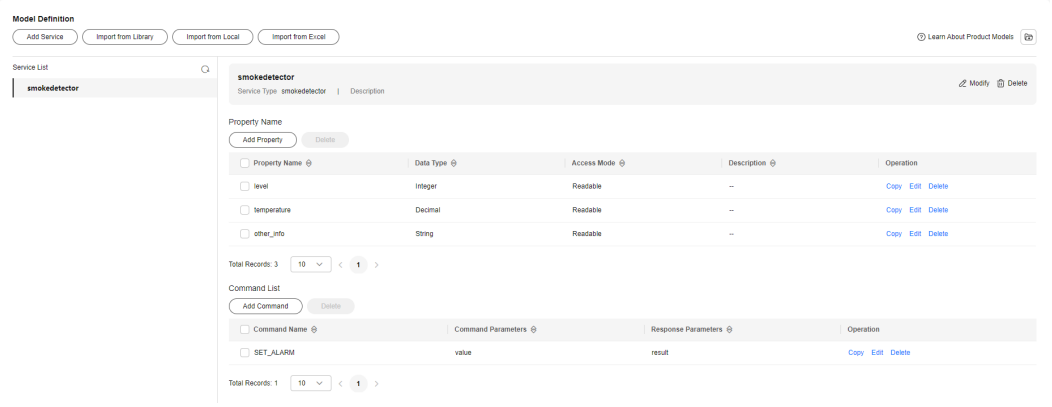
Codec for Arrays and Variable-Length Arrays

If the smoke detector needs to report the description information in arrays or variable-length arrays, perform the following steps to create messages:

Product Model

Define the product model on the product details page of the smoke detector.

Figure 3-52 Model definition - smokedetector carrying other_info



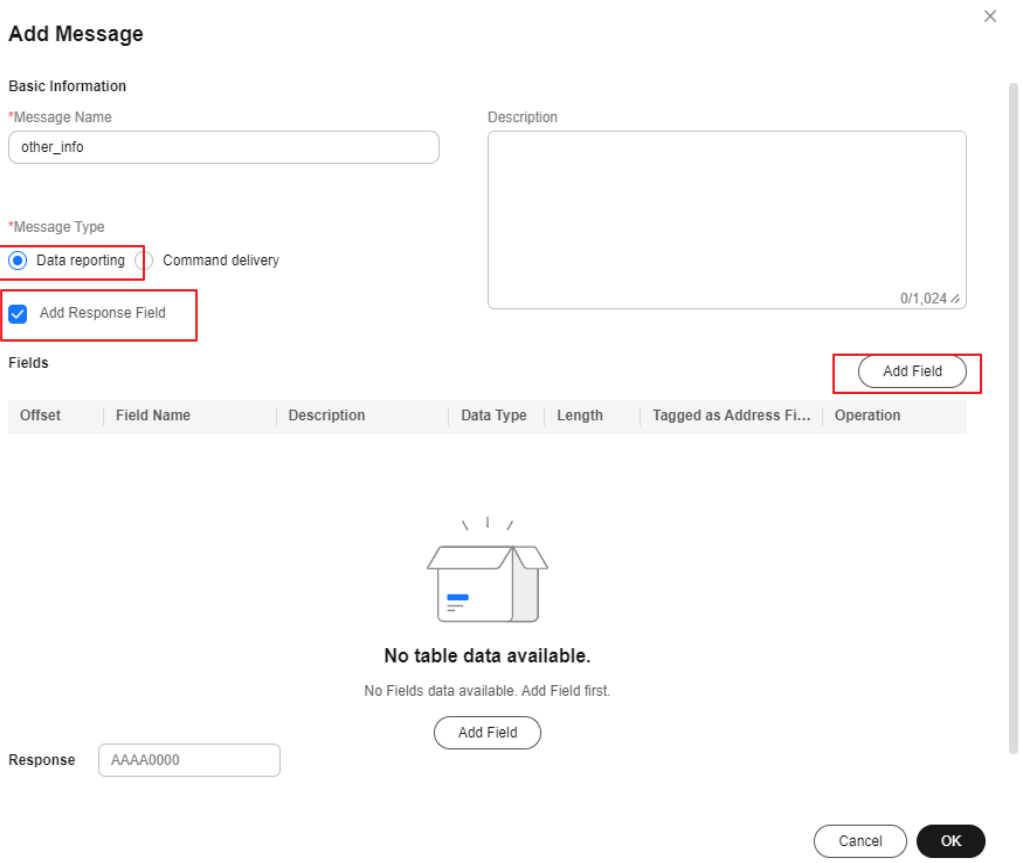
Developing a Codec

Step 1 On the smoke detector details page, click the **Codec Development** tab and click **Develop Codec**.

Step 2 Click **Add Message** to add the **other_info** message and report the description of the array type. This step is performed to decode the array binary code stream message uploaded by the device to the JSON format so that the platform can understand the message. The following is a configuration example:

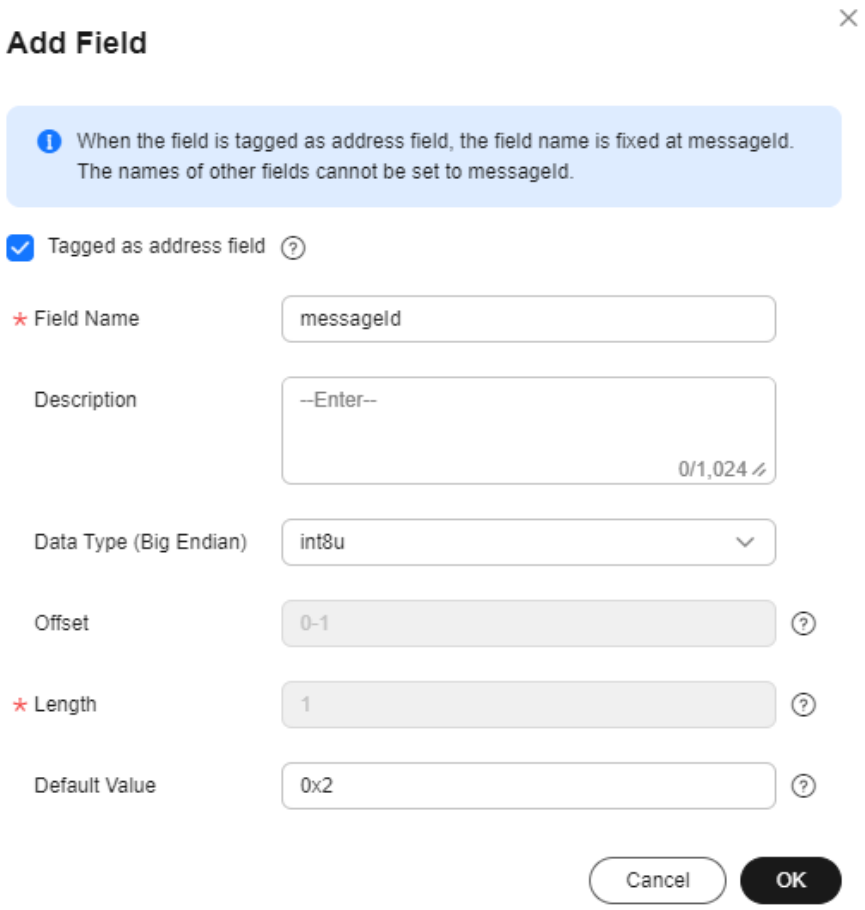
- **Message Name**: **other_info**
- **Message Type**: **Data reporting**
- **Add Response Field**: selected. After response fields are added, the platform delivers the response data set by the application to the device after receiving the data reported by the device.
- Response: AAAA0000 (default)

Figure 3-53 Adding a message - other_info



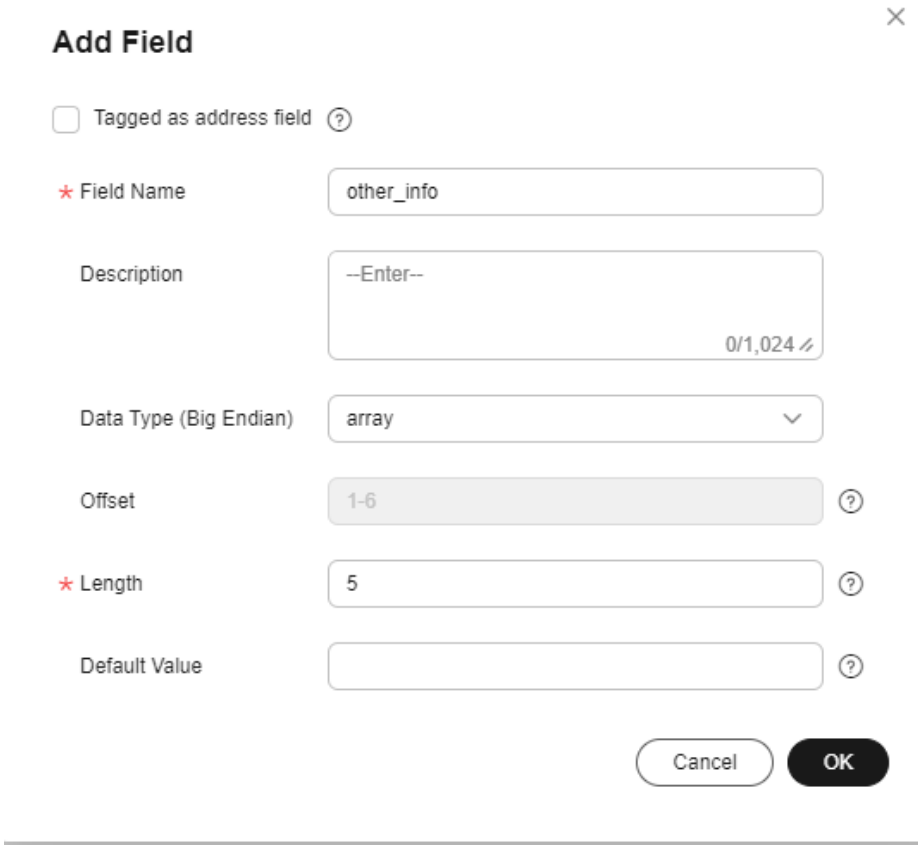
1. Click **Add Field** to add the **messageId** field, which indicates the message type. In this scenario, the value **0x0** is used to identify the message that reports the fire severity and temperature, **0x1** is used to identify the message that reports only the temperature, and **0x2** is used to identify the message that reports the description (of the array type). For details about the message ID, data type, length, default value, and offset, see 1.

Figure 3-54 Adding a field - messageId (0x2)



2. Add the **other_info** field and set **Data Type** to **array**, which indicates the description of the array type. In this scenario, set **Length** to **5**. For details about the field name, default value, and offset, see 2.

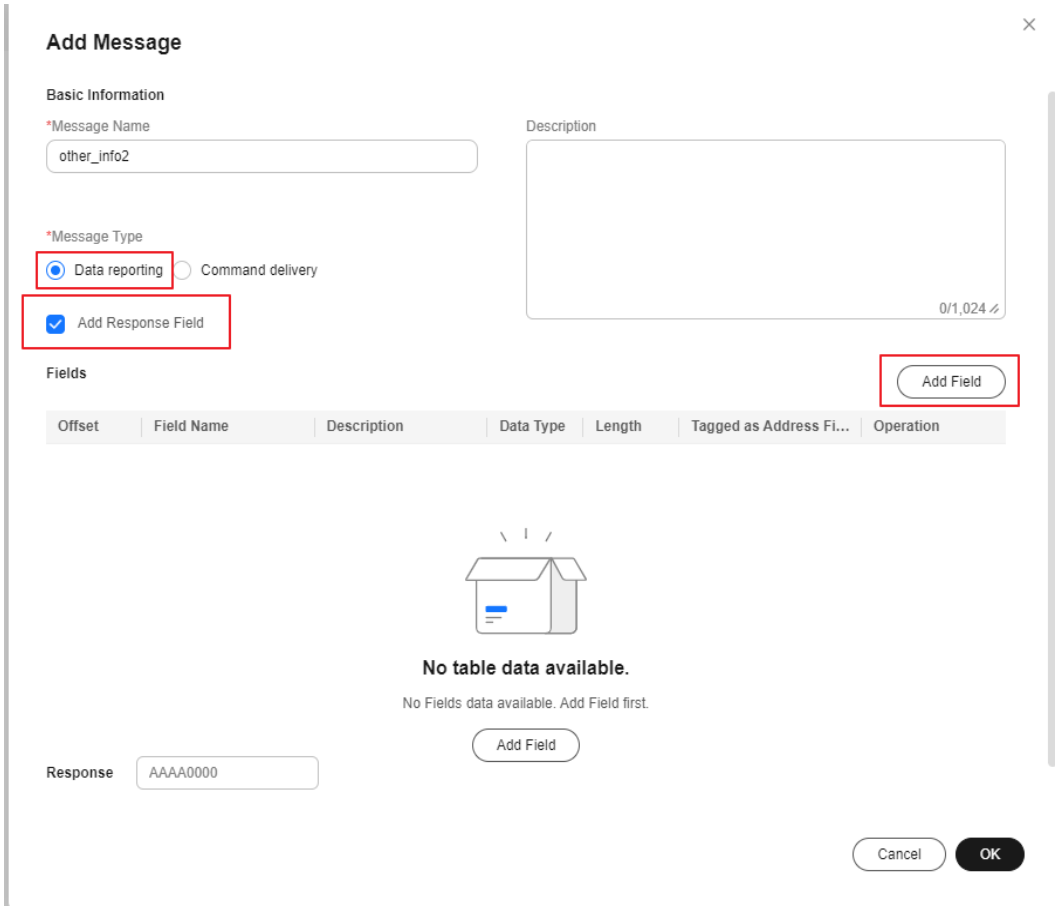
Figure 3-55 Adding a field - other_info as array



Step 3 Click **Add Message** to add the **other_info2** message and report the description of the variable-length array type. This step is performed to decode the binary code stream message of variable-length arrays uploaded by the device to the JSON format so that the platform can understand the message. The following is a configuration example:

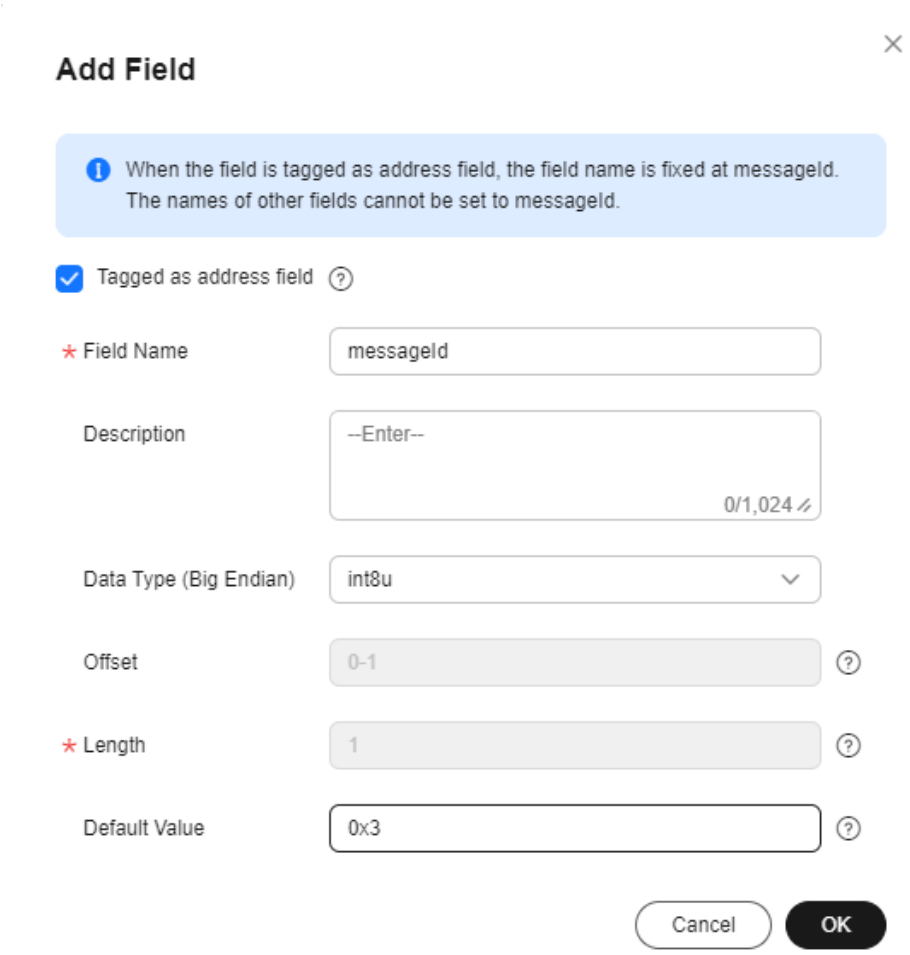
- **Message Name**: other_info2
- **Message Type**: Data reporting
- **Add Response Field**: selected. After response fields are added, the platform delivers the response data set by the application to the device after receiving the data reported by the device.
- **Response**: AAAA0000 (default)

Figure 3-56 Adding a message - other_info2



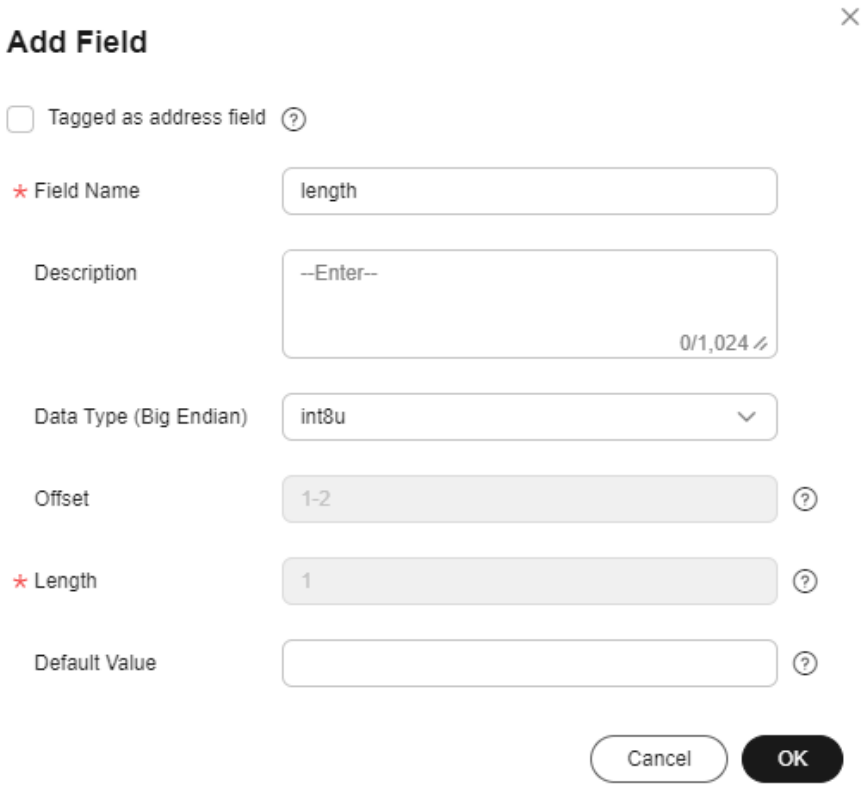
1. Click **Add Field** to add the **messageId** field, which indicates the message type. In this scenario, the value **0x0** is used to identify the message that reports the fire severity and temperature, **0x1** is used to identify the message that reports only the temperature, and **0x3** is used to identify the message that reports the description (of the variable-length array type). For details about the message ID, data type, length, default value, and offset, see **1**.

Figure 3-57 Adding a field - messageId (0x3)



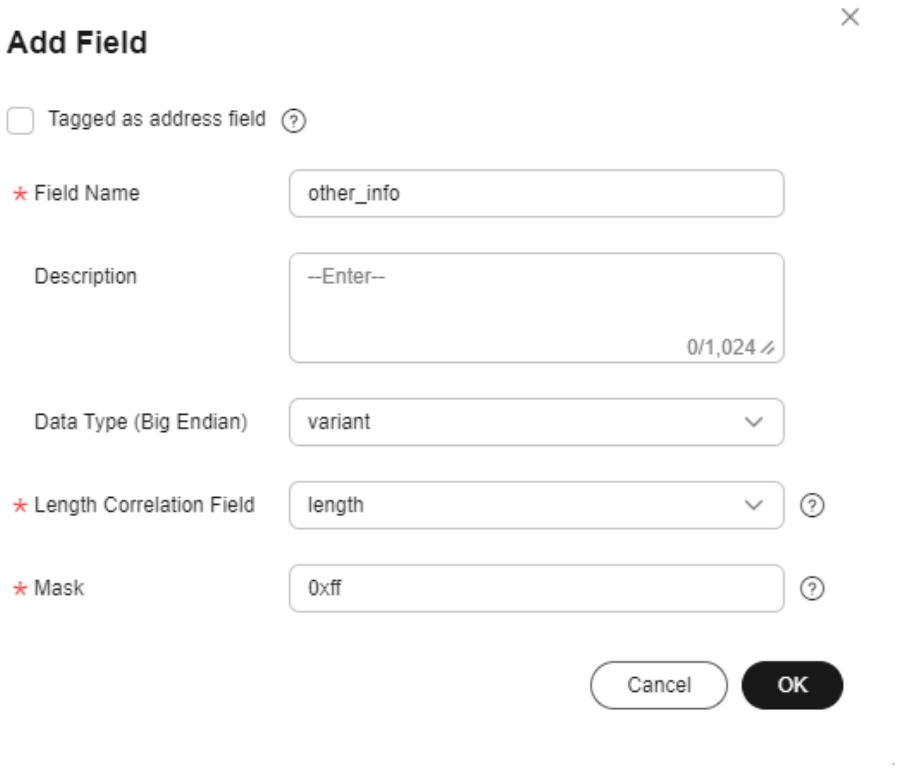
2. Add the **length** field to indicate the length of an array. **Data Type** is configured based on the length of the variable-length array. If the array contains 255 or fewer characters, set this parameter to **int8u**. For details about the length, default value, and offset, see 2.

Figure 3-58 Adding a field - length



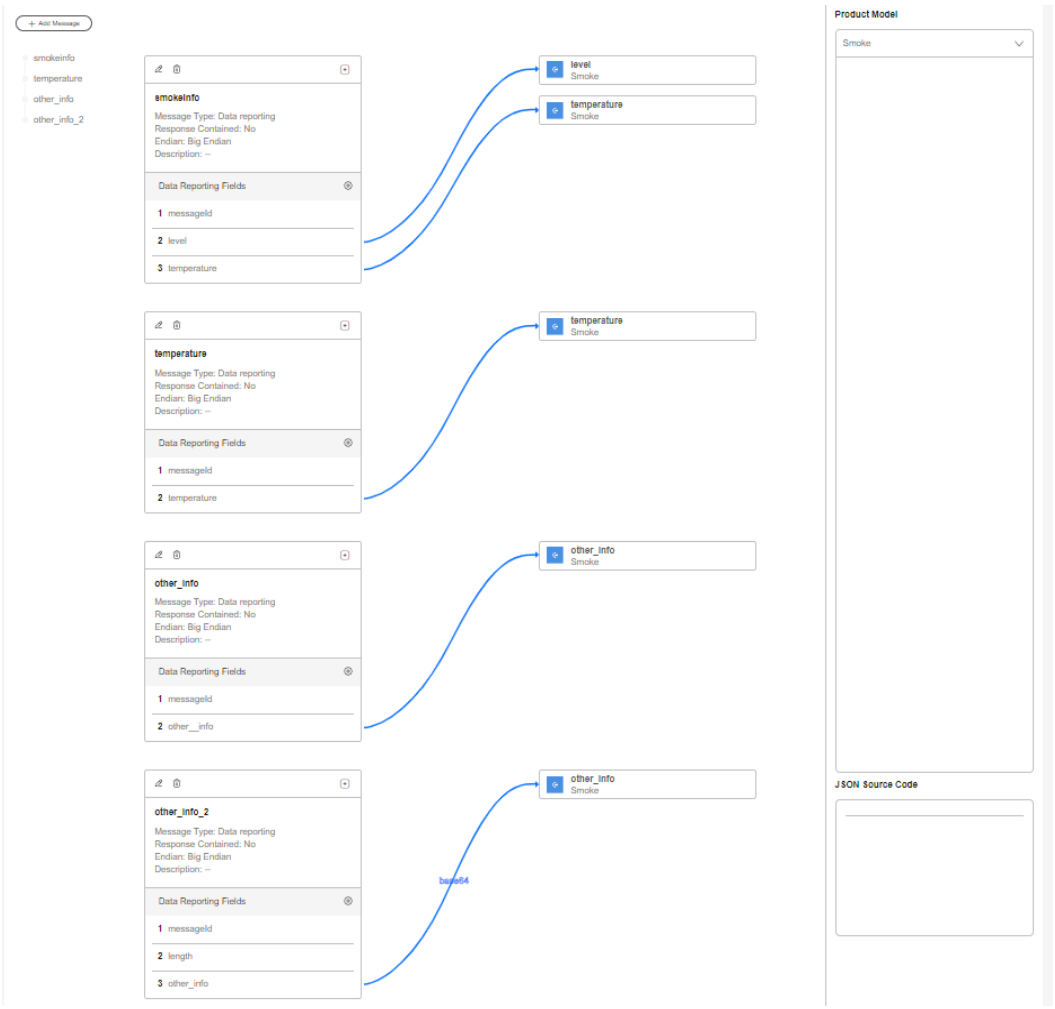
3. Add the **other_info** field and set **Data Type** to **variant**, which indicates the description of the variable-length array type. Set **Length Correlation Field** to **length**, indicating that the length of the current variable-length array is determined by the reported value of length. The default mask is **0xff**, which is used to calculate the actual length of the array. For example, if the value of **Length Correlation Field** is 5, the binary value is **00000101**. If the mask is **0xff**, the binary value is **11111111**. The result of the AND operation on these two values is **00000101**, that is, **5** in decimal format. Therefore, the length of this array that takes effect is 5 bytes. For example, if the reported data is **03051234567890**, its message ID is **03**, its length is 5 bytes, and the code stream corresponding to **other_info** is **1234567890**.

Figure 3-59 Adding a field - other_info as variant



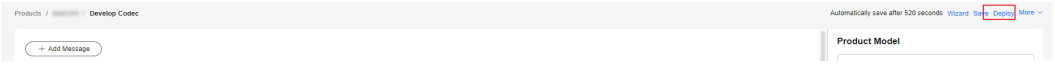
Step 4 Drag the property fields in **Device Model** on the right to set up a mapping between the corresponding fields in the data reporting messages.

Figure 3-60 Developing a codec - Data reporting field mapping (other_info to variant)



Step 5 Click **Save** and then **Deploy** to deploy the codec on the platform.

Figure 3-61 Deploying a codec



----End

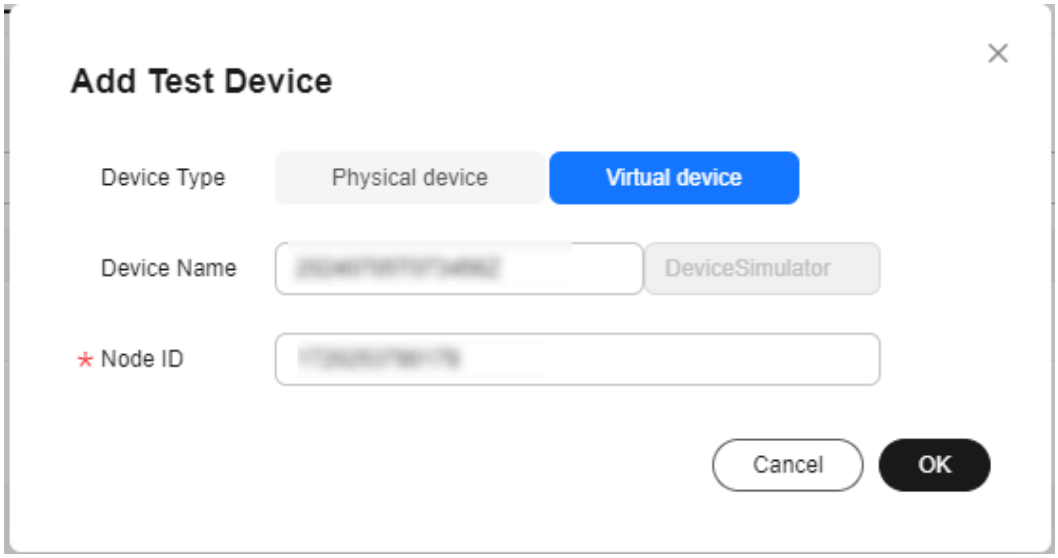
Testing the Codec

Step 1 On the product details page of the smoke detector, click the **Online Debugging** tab and click **Add Test Device**.

Step 2 You can use a real device or virtual device for debugging based on your service scenario. For details, see **Online Debugging**. The following uses a virtual device as an example to describe how to debug a codec.

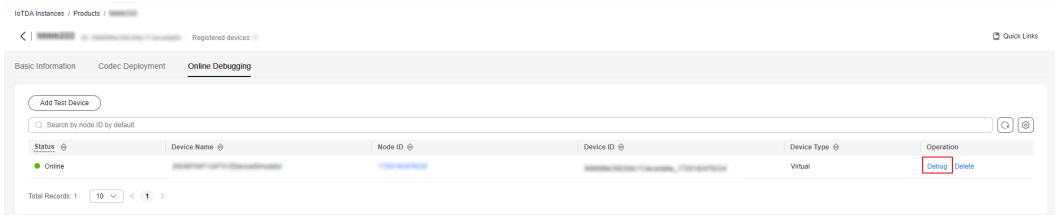
In the **Add Test Device** dialog box, select **Virtual device** for **Device Type** and click **OK**. The virtual device name contains **DeviceSimulator**. Only one virtual device can be created for each product.

Figure 3-62 Online debugging - Creating a virtual device



Step 3 Click **Debug** to access the debugging page.

Figure 3-63 Entering debugging

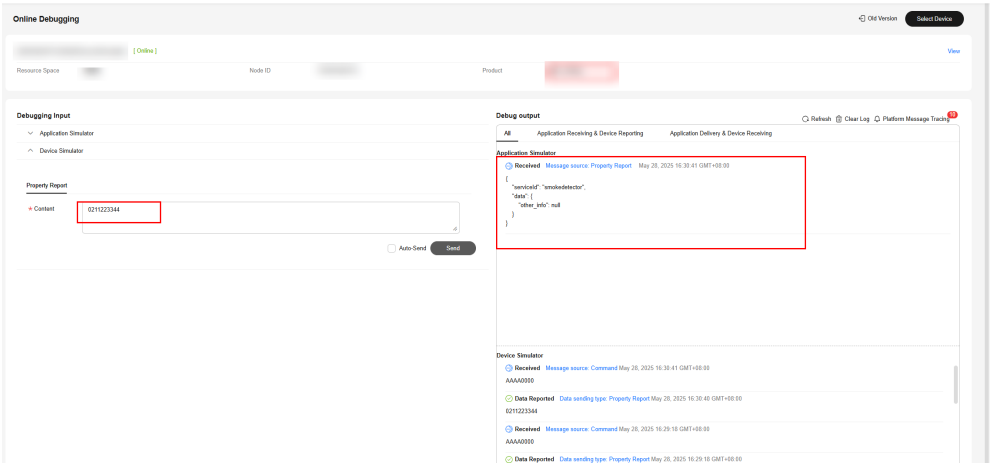


Step 4 Use the device simulator to report the description of the array type.

For example, a hexadecimal code stream (0211223344) is reported. In this code stream, **02** indicates the **messageId** field and specifies that this message reports the description of the array type. **11223344** indicates the description and its length is four bytes.

View the data reporting result ({other_info=null}) in **Application Simulator**. The length of the description is less than five bytes. Therefore, the codec cannot parse the description.

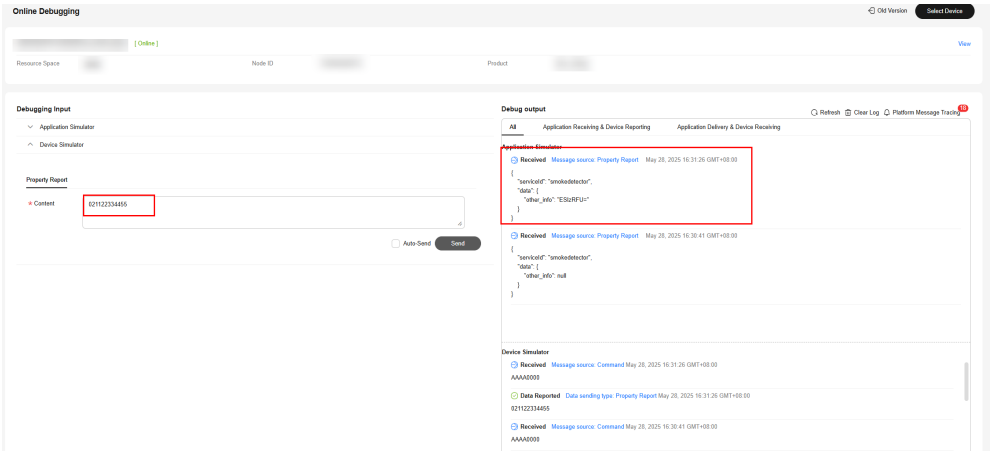
Figure 3-64 Simulating data reporting - other_info as array 1



In the hexadecimal code stream example (021122334455), **02** indicates the **messageId** field and specifies that this message reports the description of the array type. **1122334455** indicates the description and its length is five bytes.

View the data reporting result ({serviceId: smokedetector, data: {"other_info":"ESIzRFU="}}) in **Application Simulator**. The length of the description is five bytes. The description is parsed successfully by the codec.

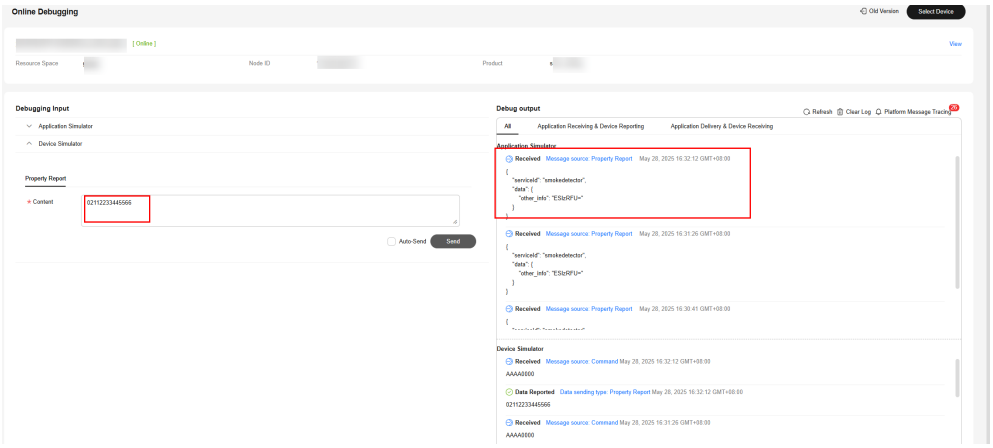
Figure 3-65 Simulating data reporting - other_info as array 2



In the hexadecimal code stream example (02112233445566), **02** indicates the **messageId** field and specifies that this message reports the description of the array type. **112233445566** indicates the description and its length is six bytes.

View the data reporting result ({serviceId: smokedetector, data: {"other_info":"ESIzRFU="}}) in **Application Simulator**. The length of the description exceeds six bytes. Therefore, the first five bytes are intercepted and parsed by the codec.

Figure 3-66 Simulating data reporting - other_info as array 3

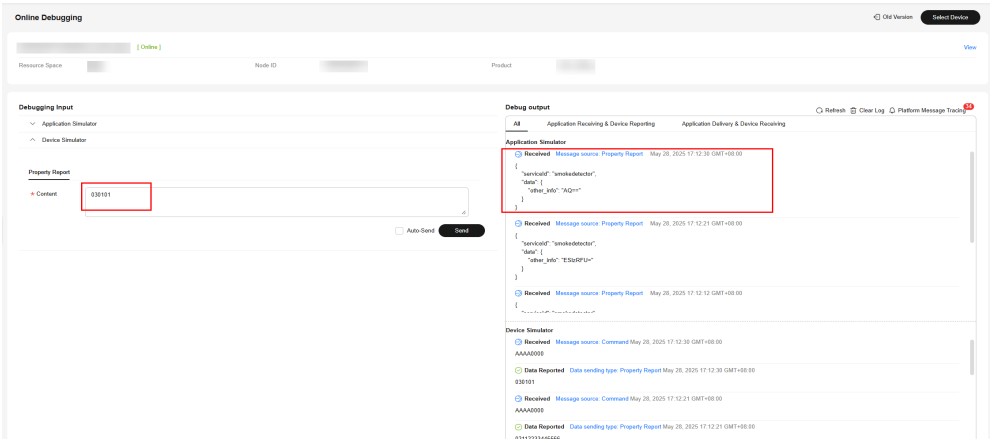


Step 5 Use the device simulator to report the description of the variable-length array type.

In the hexadecimal code stream example (030101), **03** indicates the **messageId** field and specifies that this message reports the description of the variable-length array type. The first **01** indicates the length of the description (one byte) and its length is one byte. The second **01** indicates the description and its length is one byte.

View the data reporting result ({serviceId: smokedetector, data: {"other_info":"AQ=="}}) in **Application Simulator**. **AQ==** is the encoded value of **01** using the Base64 encoding mode.

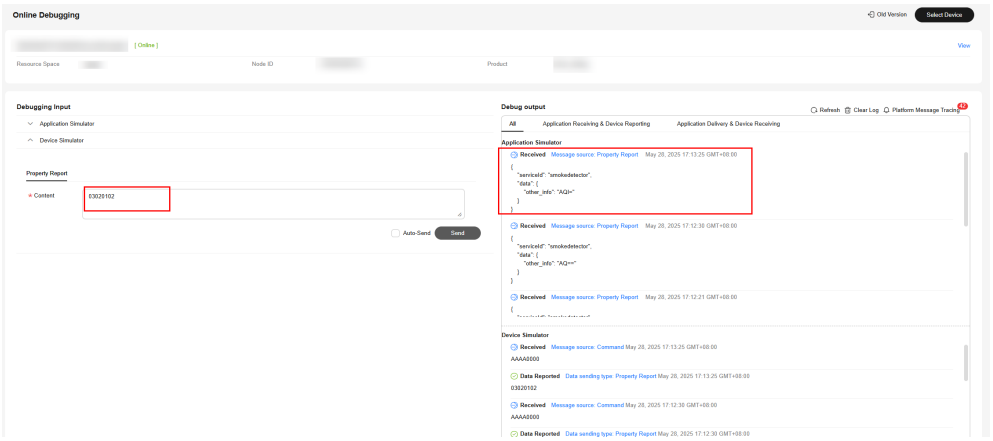
Figure 3-67 Simulating data reporting - other_info as variable-length array 1



In the hexadecimal code stream example (03020102), **03** indicates the **messageId** field and specifies that this message reports the description of the variable-length array type. **02** indicates the length of the description (two bytes) and its length is one byte. **0102** indicates the description and its length is two bytes.

View the data reporting result ({serviceId: smokedetector, data: {"other_info":"AQI="}}) in **Application Simulator**. **AQI=** is the encoded value of **01** using the Base64 encoding mode.

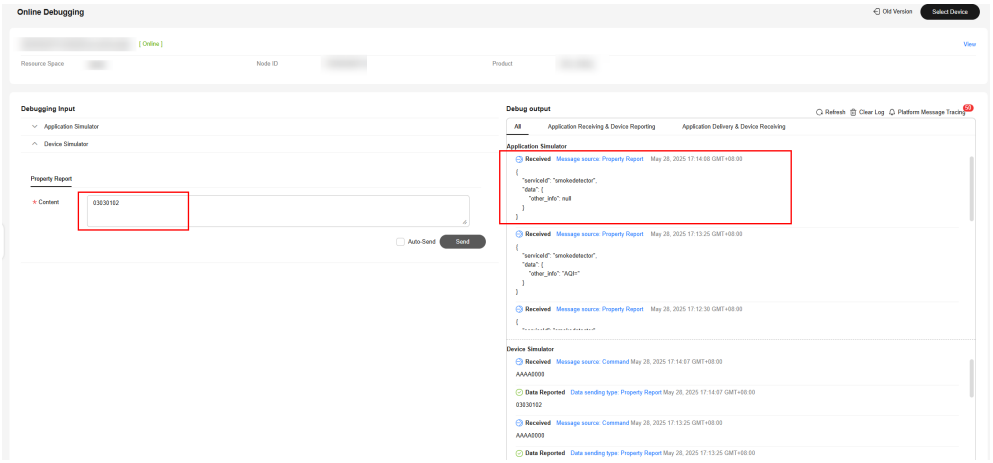
Figure 3-68 Simulating data reporting - other_info as variable-length array 2



In the hexadecimal code stream example (03030102), **03** indicates the **messageId** field and specifies that this message reports the description of the variable-length array type. The second **03** indicates the length of the description (three bytes) and its length is one byte. **0102** indicates the description and its length is two bytes.

View the data reporting result ({other_info=null}) in **Application Simulator**. The length of the description is less than three bytes. The codec fails to parse the description.

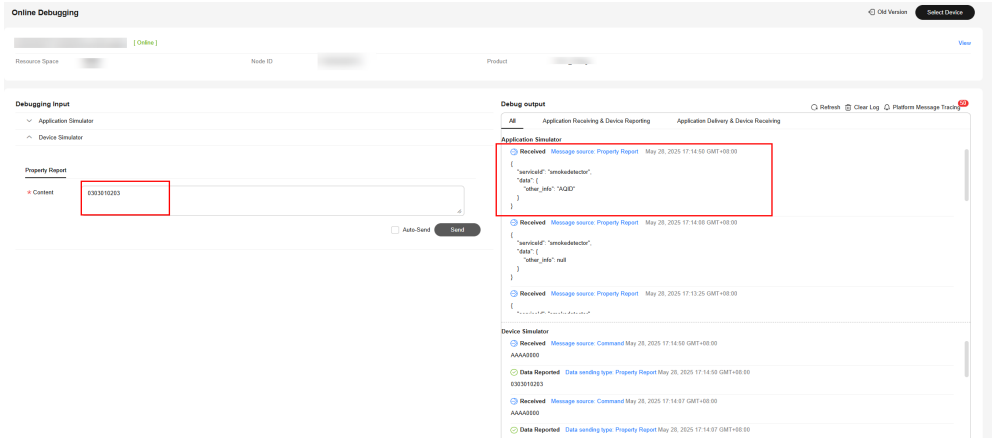
Figure 3-69 Simulating data reporting - other_info as variable-length array 3



In the hexadecimal code stream example (0303010203), **03** indicates the **messageId** field and specifies that this message reports the description of the variable-length array type. The second **03** indicates the length of the description (three bytes) and its length is one byte. **010203** indicates the description and its length is three bytes.

View the data reporting result ({serviceId: smokedetector, data: {"other_info":"AQID"}}) in **Application Simulator**. **AQID** is the encoded value of **010203** using the Base64 encoding mode.

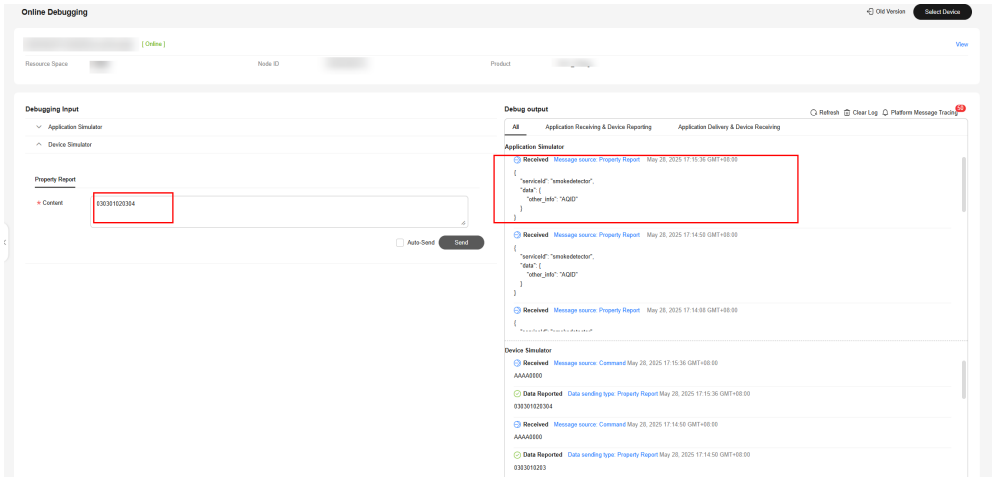
Figure 3-70 Simulating data reporting - other_info as variable-length array 4



In the hexadecimal code stream example (030301020304), **03** indicates the **messageId** field and specifies that this message reports the description of the variable-length array type. The second **03** indicates the length of the description (three bytes) and its length is one byte. **01020304** indicates the description and its length is four bytes.

View the data reporting result ({other_info=AQID}) in **Application Simulator**. The length of the description exceeds three bytes. Therefore, the first three bytes are intercepted and parsed. **AQID** is the encoded value of **010203** using the Base64 encoding mode.

Figure 3-71 Simulating data reporting - other_info as variable-length array 5



----End

Description of Base64 Encoding Modes

In Base64 encoding mode, three 8-bit bytes (3 x 8 = 24) are converted into four 6-bit bytes (4 x 6 = 24), and 00 are added before each 6-bit byte to form four 8-bit bytes. If the code stream to be encoded contains less than three bytes, fill the code stream with 0 at the end. The byte that is filled with 0 is displayed as an equal sign (=) after it is encoded.

Developers can encode hexadecimal code streams as characters or values using the Base64 encoding modes. The encoding results obtained in the two modes are different. The following uses the hexadecimal code stream 01 as an example:

- Use 01 as the characters. 01 contains fewer than three characters. Therefore, add one 0 to obtain 010. Query the ASCII code table to convert the characters into an 8-bit binary number, that is, 0 is converted into 00110000 and 1 into 00110001. Therefore, 010 can be converted into 001100000011000100110000 (3 x 8 = 24). The binary number can be split into four 6-bit numbers: 001100, 000011, 000100, and 110000. Then, pad each 6-bit number with 00 to obtain the following numbers: 00001100, 00000011, 00000100, and 00110000. The decimal numbers corresponding to the four 8-bit numbers are 12, 3, 4, and 48, respectively. You can obtain M (12), D (3), and E (4) by querying the Base64 coding table. As the last character of 010 is obtained by adding 0, the fourth 8-bit number is represented by an equal sign (=). Finally, **MDE=** is obtained by using **01** as characters.
- Use 01 as a value (that is, 1). It contains fewer than three characters. Therefore, add 00 to obtain 100. Convert 100 into an 8-bit binary number, that is, 0 is converted into 00000000 and 1 is converted into 00000001. Therefore, 100 can be converted into 000000010000000000000000 (3 x 8 = 24). The binary number can be split into four 6-bit numbers: 000000, 010000, 000000, and 000000. Then, pad each 6-bit number with 00 to obtain 00000000, 00010000, 00000000, and 00000000. The decimal numbers corresponding to the four 8-bit numbers are 0, 16, 0, and 0, respectively. You can obtain A (0) and Q (16) by querying the Base64 coding table. As the last two characters of 100 are obtained by adding 0, the third and fourth 8-bit numbers are represented by two equal signs (==). Finally, **AQ==** is obtained by using **01** as a value.

Summary

- When the data is an array or a variable-length array, the codec encodes and decodes the data using Base64. For data reporting messages, the hexadecimal code streams are encoded using Base64. For example, **01** is encoded into **AQ==**. For command delivery messages, characters are decoded using Base64. For example, **AQ==** is decoded to **01**.
- When the data type of a field is **variant** (variable-length array type), the field must be associated with the **length** field. The data type of the **length** field must be **int**.
- For variable-length arrays, the codecs for command delivery and data reporting are developed in the same way.
- When the codecs that are developed online encode data using Base64, hexadecimal code streams are encoded as **values**.

3.2.4.3 JavaScript Script-based Development

The IoT platform can encode and decode JavaScript scripts. Based on the script files you submit, the IoT platform can convert between binary and JSON formats as well as between different JSON formats. This topic uses a smoke detector as an example to describe how to develop a JavaScript codec that supports device property reporting and command delivery, and describes the format conversion requirements and debugging method of the codec.

NOTE

After December 1, 2024, JavaScript-based codec development is no longer available on the platform for new users. You are advised to use FunctionGraph to write JavaScript scripts. For details, see **Overview**.

NOTE

- JavaScript syntax rules must comply with **ECMAScript 5.1 specifications**.
- The codec script supports only **let** and **const** of ECMAScript 6. Other expressions, such as the arrow function, are not supported.
- The size of a JavaScript script cannot exceed 1 MB.
- After the JavaScript script is deployed on a product, the JavaScript script parses upstream and downstream data of all devices under the product. When you develop a JavaScript codec, take all upstream and downstream scenarios into consideration.
- The JSON upstream data obtained after being decoded by the JavaScript codec must meet the format requirements of the platform. For details about the format requirements, see **Data Decoding Format Definition**.
- For the JSON format definition of downstream commands, see **Data Encoding Format Definition**. If the JavaScript codec is used for encoding, the JSON format of the platform must be converted into the corresponding binary code stream or another JSON format.
- You can select the auto save option in the upper right corner of the script text box to let the system automatically save the scripts every 10 seconds.

Defining a Smoke Detector

Scenario

A smoke detector provides the following functions:

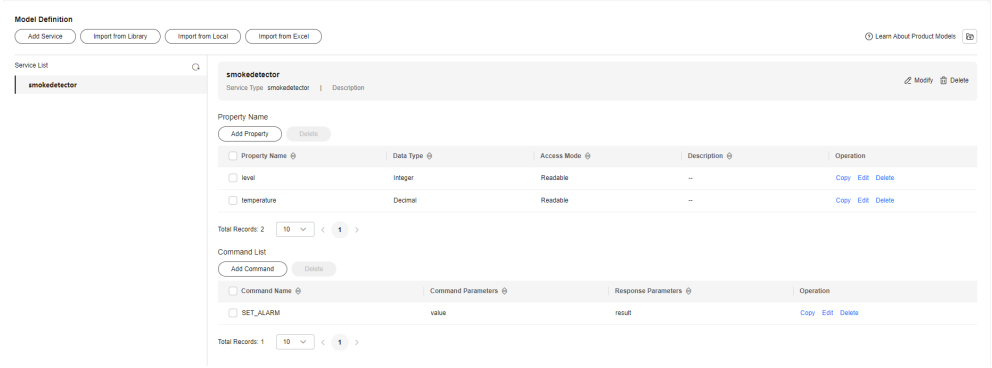
- Reporting smoke alarms (fire severity) and temperature.
- Receiving and running remote control commands, which can be used to enable the alarm function remotely. For example, the smoke detector can report the temperature on the fire scene and remotely trigger a smoke alarm for evacuation.
- The smoke detector has weak capabilities and cannot report data in JSON format defined by the device APIs, but reporting simple binary data.

Product Model

Define the **product model** on the product details page of the smoke detector.

- **level**: indicates the fire severity.
- **temperature**: indicates the temperature at the fire scene.
- **SET_ALARM**: indicates whether to enable or disable the alarm function. The value **0** indicates that the alarm function is disabled, and **1** indicates that the alarm function is enabled.

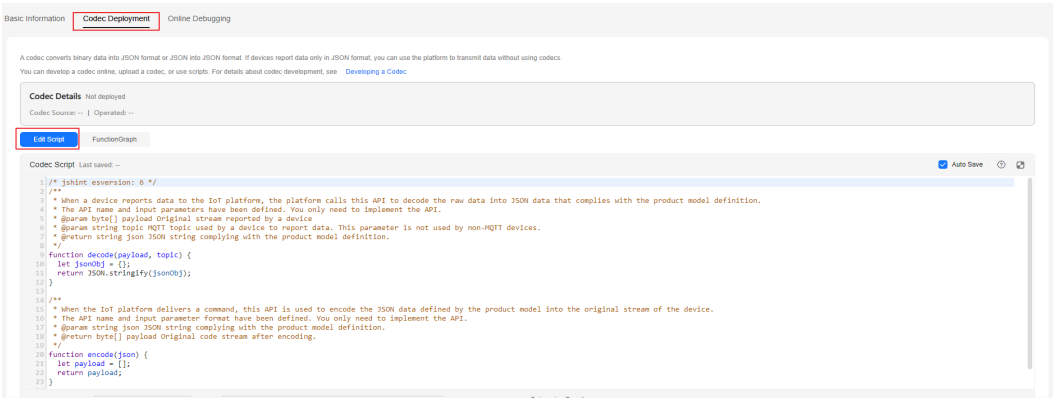
Figure 3-72 Model definition - smokedetector



Developing a Codec

Step 1 On the smoke detector details page, click the **Codec Development** tab and click **Edit Script**.

Figure 3-73 Developing a codec - Script-based development



Step 2 Write a script to convert binary data into JSON data. The script must implement the following methods:

- Decode: Converts the binary data reported by a device into the JSON format defined in the product model. For details about the JSON format requirements, see **Data Decoding Format Definition**.
- Encode: Converts JSON data into binary data supported by the device when the platform sends downstream data to the device. For details about the JSON format requirements, see **Data Encoding Format Definition**.

The following is an example of JavaScript implemented for the current smoke detector:

```
// Upstream message types
var MSG_TYPE_PROPERTIES_REPORT = 'properties_report'; // Device property reporting
var MSG_TYPE_COMMAND_RSP = 'command_response'; // Command response
var MSG_TYPE_PROPERTIES_SET_RSP = 'properties_set_response'; // Property setting response
var MSG_TYPE_PROPERTIES_GET_RSP = 'properties_get_response'; // Property query response
var MSG_TYPE_MESSAGE_UP = 'message_up'; // Device message reporting
// Downstream message types
var MSG_TYPE_COMMANDS = 'commands'; // Command delivery
var MSG_TYPE_PROPERTIES_SET = 'properties_set'; // Property setting request
var MSG_TYPE_PROPERTIES_GET = 'properties_get'; // Property query request
var MSG_TYPE_MESSAGE_DOWN = 'messages'; // Platform message delivery
// Mapping between topics and upstream MQTT message types
var TOPIC_REG_EXP = {
```



```
    'properties_report': new RegExp('\\$oc/devices/(\\S+)/sys/properties/report'),
    'properties_set_response': new RegExp('\\$oc/devices/(\\S+)/sys/properties/set/response/request_id=(\\S
+)'),
    'properties_get_response': new RegExp('\\$oc/devices/(\\S+)/sys/properties/get/response/request_id=(\\S
+)'),
    'command_response': new RegExp('\\$oc/devices/(\\S+)/sys/commands/response/request_id=(\\S+)'),
    'message_up': new RegExp('\\$oc/devices/(\\S+)/sys/messages/up')
};
/*
Example: When a smoke detector reports properties and returns a command response, it uses binary code
streams. The JavaScript script will decode the binary code streams into JSON data that complies with the
product model definition.
Input parameters:
  payload:[0x00, 0x50, 0x00, 0x5a]
  topic:$oc/devices/cf40f3c4-7152-41c6-a201-a2333122054a/sys/properties/report
Output:
  {"msg_type":"properties_report","services":[{"service_id":"smokerdector","properties":
{"level":80,"temperature":90}}]}
Input parameters:
  payload: [0x02, 0x00, 0x00, 0x01]
  topic: $oc/devices/cf40f3c4-7152-41c6-a201-a2333122054a/sys/commands/response/
request_id=bf40f0c4-4022-41c6-a201-c5133122054a
Output:

{"msg_type":"command_response","result_code":0,"command_name":"SET_ALARM","service_id":"smokerdect
or","paras":{"value":"1"}}
*/
function decode(payload, topic) {
    var jsonObj = {};
    var msgType = '';
   // Parse the message type based on the topic parameter, if available.
    if (null != topic) {
        msgType = topicParse(topic);
    }
    // Perform the AND operation on the payload by using 0xFF to obtain the corresponding complementary
code.
    var uint8Array = new Uint8Array(payload.length);
    for (var i = 0; i < payload.length; i++) {
        uint8Array[i] = payload[i] & 0xff;
    }
    var dataView = new DataView(uint8Array.buffer, 0);
    // Convert binary data into the format used for property reporting.
    if (msgType == MSG_TYPE_PROPERTIES_REPORT) {
        // Set the value of serviceId, which corresponds to smokerdector in the product model.
        var serviceId = 'smokerdector';
        // Obtain the level value from the code stream.
        var level = dataView.getInt16(0);
        // Obtain the temperature value from the code stream.
        var temperature  = dataView.getInt16(2);
        // Convert the data to the JSON format used by property reporting.
        jsonObj = {"msg_type":"properties_report","services":[{"service_id":serviceId,"properties":
{"level":level,"temperature":temperature}}]};
    }else if (msgType == MSG_TYPE_COMMAND_RSP) { // Convert binary data into the format used by a
command response.
        // Set the value of serviceId, which corresponds to smokerdector in the product model.
        var serviceId = 'smokerdector';
        var command = dataView.getInt8(0); // Obtain the command name ID from the binary code stream.
        var command_name = '';
        if (2 == command) {
            command_name = 'SET_ALARM';
        }
        var result_code  = dataView.getInt16(1); // Obtain the command execution result from the binary code
stream.
        var value = dataView.getInt8(3); // Obtain the returned value of the command execution result from
the binary code stream.
       // Convert data to the JSON format used by the command response.
        jsonObj =
{"msg_type":"command_response","result_code":result_code,"command_name":command_name,"service_id":
serviceId,"paras":{"value":value}};
```



```
    }
    // Convert data into a string in JSON format.
    return JSON.stringify(jsonObj);
}
/*
Sample data: When a command is delivered, data in JSON format on IoTDA is encoded into a binary code stream using the encode method of JavaScript.
Input parameters ->
    {"msg_type":"commands","command_name":"SET_ALARM","service_id":"smokerdector","paras":{"value":1}}
Output ->
    [0x01,0x00, 0x00, 0x01]
*/
function encode(json) {
    // Convert data to a JSON object.
    var jsonObj = JSON.parse(json);
   // Obtain the message type.
    var msgType = jsonObj.msg_type;
    var payload = [];
   // Convert data in JSON format to binary data.
    if (msgType == MSG_TYPE_COMMANDS) // Command delivery
    {
        payload = payload.concat(buffer_uint8(1)); // Identify command delivery.
        if (jsonObj.command_name == 'SET_ALARM') {
            payload = payload.concat(buffer_uint8(0)); // Command name
        }
        var paras_value = jsonObj.paras.value;
        payload = payload.concat(buffer_int16(paras_value)); // Set the command property value.
    }
   // Return the encoded binary data.
    return payload;
}
// Parse the message type based on the topic name.
function topicParse(topic) {
    for(var type in TOPIC_REG_EXP){
        var pattern = TOPIC_REG_EXP[type];
        if (pattern.test(topic)) {
            return type;
        }
    }
    return '';
}
// Convert an 8-bit unsigned integer into a byte array.
function buffer_uint8(value) {
    var uint8Array = new Uint8Array(1);
    var dataView = new DataView(uint8Array.buffer);
    dataView.setUint8(0, value);
    return [].slice.call(uint8Array);
}
// Convert a 16-bit unsigned integer into a byte array.
function buffer_int16(value) {
    var uint8Array = new Uint8Array(2);
    var dataView = new DataView(uint8Array.buffer);
    dataView.setInt16(0, value);
    return [].slice.call(uint8Array);
}
// Convert a 32-bit unsigned integer into a byte array.
function buffer_int32(value) {
    var uint8Array = new Uint8Array(4);
    var dataView = new DataView(uint8Array.buffer);
    dataView.setInt32(0, value);
    return [].slice.call(uint8Array);
}
```

Step 3 Debug the script online. After the script is edited, select the simulation type and enter the simulation data to debug the script online.

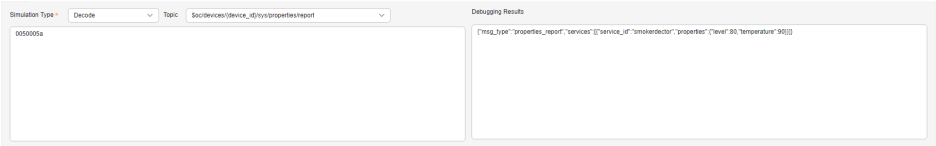
1. Use the simulation device to convert binary code streams into JSON data when reporting property data.

- Select the topic used by device property reporting: $oc/devices/{device_id}/sys/properties/report.
- Select **Decode** for **Simulation Type**, enter the following simulated device data, and click **Debug**.

```
0050005a
```

- The script codec engine converts binary code streams into the JSON format based on input parameters and the decode method in the submitted JavaScript script, and displays the debugging result in the text box.

Figure 3-74 Script-based development - Debugging and decoding



- Check whether the debugging result meets the expectation. If the debugging result does not meet the expectation, modify the code and perform debugging again.

2. Convert a command delivered by an application into binary code streams that can be identified by the device.
   - Select **Encode** for **Simulation Type**, enter the command delivery format to be simulated, and click **Debug**.

```
{
   "msg_type": "commands",
   "request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
   "command_name": "SET_ALARM",
   "service_id": "smokerdector",
   "paras": {
      "value": "1"
   }
}
```

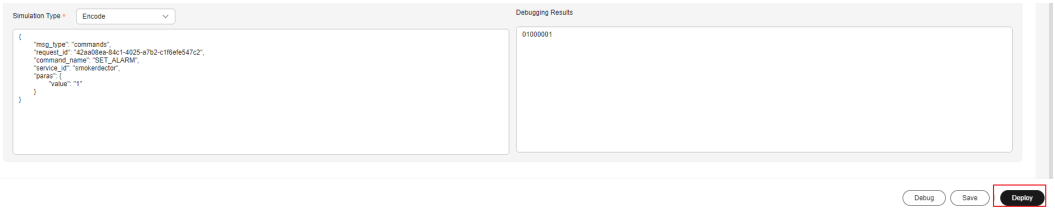
   - The script codec engine converts JSON data into the binary code streams based on input parameters and the encode method in the submitted JavaScript script, and displays the debugging result in the text box.

Figure 3-75 Script-based development - Debugging and coding

   - Check whether the debugging result meets the expectation. If the debugging result does not meet the expectation, modify the code and perform debugging again.

Step 4 Deploy the script. After confirming that the script can be correctly encoded and decoded, click **Deploy** to submit the script to the IoT platform so that the IoT platform can invoke the script when data is sent and received.

Figure 3-76 Script-based development - Deployment



Step 5 Use a physical device for online debugging. Before using the script, use a real device to communicate with the IoT platform to verify that the IoT platform can invoke the script and parse upstream and downstream data.

----End

JavaScript Codec Template

The following is an example of the JavaScript codec template. Developers need to implement the corresponding API based on the template provided by the platform.

```
/**
* When a device reports data to the IoT platform, the IoT platform calls this API to decode the raw data of the device into JSON data that complies with the product model definition.
* The API name and input parameters have been defined. You only need to implement the API.
* @param byte[] payload    Original code stream reported by the device
* @param string topic      Topic to which an MQTT device reports data. This parameter is not carried when a non-MQTT device reports data.
* @return string json      JSON character string that complies with the product model definition
 */
function decode(payload, topic) {
    var jsonObj = {};
    return JSON.stringify(jsonObj);
}

/**
* When the IoT platform delivers a command, it calls this API to encode the JSON data defined in the product model into the original code stream of the device.
* The API name and input parameter format have been defined. You only need to implement the API.
* @param string json      JSON character string that complies with the product model definition
* @return byte[] payload   Original code stream after being encoded
 */
function encode(json) {
    var payload = [];
    return payload;
}
```

JavaScript Codec Example for MQTT Device Access

The following is an example of JavaScript codec of MQTT devices. You can convert the binary format to the JSON format in the corresponding scenario based on the example.

```
// Upstream message types
var MSG_TYPE_PROPERTIES_REPORT = 'properties_report'; // Device property reporting
var MSG_TYPE_COMMAND_RSP = 'command_response'; // Command response
var MSG_TYPE_PROPERTIES_SET_RSP = 'properties_set_response'; // Property setting response
var MSG_TYPE_PROPERTIES_GET_RSP = 'properties_get_response'; // Property query response
var MSG_TYPE_MESSAGE_UP = 'message_up'; // Device message reporting
// Downstream message types
var MSG_TYPE_COMMANDS = 'commands'; // Command delivery
var MSG_TYPE_PROPERTIES_SET = 'properties_set'; // Property setting request
var MSG_TYPE_PROPERTIES_GET = 'properties_get'; // Property query request
var MSG_TYPE_MESSAGE_DOWN = 'messages'; // Platform message delivery
```



```
// Mapping between topics and upstream MQTT message types
var TOPIC_REG_EXP = {
    'properties_report': new RegExp('\\$oc/devices/(\\S+)/sys/properties/report'),
    'properties_set_response': new RegExp('\\$oc/devices/(\\S+)/sys/properties/set/response/request_id=(\\S
+)'),
    'properties_get_response': new RegExp('\\$oc/devices/(\\S+)/sys/properties/get/response/request_id=(\\S
+)'),
    'command_response': new RegExp('\\$oc/devices/(\\S+)/sys/commands/response/request_id=(\\S+)'),
    'message_up': new RegExp('\\$oc/devices/(\\S+)/sys/messages/up')
};
/*
Example: When a smoke detector reports properties and returns a command response, it uses binary code
streams. The JavaScript script will decode the binary code streams into JSON data that complies with the
product model definition.
Input parameters:
  payload:[0x00, 0x50, 0x00, 0x5a]
  topic:$oc/devices/cf40f3c4-7152-41c6-a201-a2333122054a/sys/properties/report
Output:
  {"msg_type":"properties_report","services":[{"service_id":"smokerdector","properties":
{"level":80,"temperature":90}}]}
Input parameters:
  payload: [0x02, 0x00, 0x00, 0x01]
  topic: $oc/devices/cf40f3c4-7152-41c6-a201-a2333122054a/sys/commands/response/
request_id=bf40f0c4-4022-41c6-a201-c5133122054a
Output:

{"msg_type":"command_response","result_code":0,"command_name":"SET_ALARM","service_id":"smokerdect
or","paras":{"value":"1"}}
*/
function decode(payload, topic) {
    var jsonObj = {};
    var msgType = '';
   // Parse the message type based on the topic parameter, if available.
    if (null != topic) {
        msgType = topicParse(topic);
    }
    // Perform the AND operation on the payload by using 0xFF to obtain the corresponding complementary
code.
    var uint8Array = new Uint8Array(payload.length);
    for (var i = 0; i < payload.length; i++) {
        uint8Array[i] = payload[i] & 0xff;
    }
    var dataView = new DataView(uint8Array.buffer, 0);
    // Convert binary data into the format used for property reporting.
    if (msgType == MSG_TYPE_PROPERTIES_REPORT) {
        // Set the value of serviceId, which corresponds to smokerdector in the product model.
        var serviceId = 'smokerdector';
        // Obtain the level value from the code stream.
        var level = dataView.getInt16(0);
        // Obtain the temperature value from the code stream.
        var temperature = dataView.getInt16(2);
        // Convert the data to the JSON format used by property reporting.
        jsonObj = {
            "msg_type": "properties_report",
            "services": [{"service_id": serviceId, "properties": {"level": level, "temperature": temperature}}]
        };
    } else if (msgType == MSG_TYPE_COMMAND_RSP) { // Convert binary data into the format used by a
command response.
        // Set the value of serviceId, which corresponds to smokerdector in the product model.
        var serviceId = 'smokerdector';
        var command = dataView.getInt8(0); // Obtain the command name ID from the binary code stream.
        var command_name = '';
        if (2 == command) {
            command_name = 'SET_ALARM';
        }
        var result_code  = dataView.getInt16(1); // Obtain the command execution result from the binary code
stream.
        var value = dataView.getInt8(3); // Obtain the returned value of the command execution result from
the binary code stream.
```



```
      // Convert data to the JSON format used by the command response.
       jsonObj = {
           "msg_type": "command_response",
           "result_code": result_code,
           "command_name": command_name,
           "service_id": serviceId,
           "paras": {"value": value}
       };
   } else if (msgType == MSG_TYPE_PROPERTIES_SET_RSP) {
      // Convert data to the JSON format used by the property setting response.
       //jsonObj = {"msg_type":"properties_set_response","result_code":0,"result_desc":"success"};
   } else if (msgType == MSG_TYPE_PROPERTIES_GET_RSP) {
      // Convert data to the JSON format used by the property query response.
       //jsonObj = {"msg_type":"properties_get_response","services":[{"service_id":"analog","properties":
{"PhV_phsA":"1","PhV_phsB":"2"}}]};
   } else if (msgType == MSG_TYPE_MESSAGE_UP) {
       // Convert the data to the JSON format used by message reporting.
       //jsonObj = {"msg_type":"message_up","content":"hello"};
   }
   // Convert data into a string in JSON format.
   return JSON.stringify(jsonObj);
}
/*
Sample data: When a command is delivered, data in JSON format on IoTDA is encoded into a binary code
stream using the encode method of JavaScript.
Input parameters ->
   {"msg_type":"commands","command_name":"SET_ALARM","service_id":"smokerdector","paras":
{"value":1}}
Output ->
    [0x01,0x00, 0x00, 0x01]
*/
function encode(json) {
   // Convert data to a JSON object.
   var jsonObj = JSON.parse(json);
  // Obtain the message type.
   var msgType = jsonObj.msg_type;
   var payload = [];
  // Convert data in JSON format to binary data.
   if (msgType == MSG_TYPE_COMMANDS) { // Command delivery
      // Command delivery format example:
{"msg_type":"commands","command_name":"SET_ALARM","service_id":"smokerdector","paras":{"value":1}}
       // Convert the format used by command delivery to a binary code stream.
       payload = payload.concat(buffer_uint8(1)); // Identify command delivery.
       if (jsonObj.command_name == 'SET_ALARM') {
           payload = payload.concat(buffer_uint8(0)); // Command name.
       }
       var paras_value = jsonObj.paras.value;
       payload = payload.concat(buffer_int16(paras_value)); // Set the command property value.
   } else if (msgType == MSG_TYPE_PROPERTIES_SET) {
       // Property setting format example: {"msg_type":"properties_set","services":
[{"service_id":"Temperature","properties":{"value":57}}]}
      // Convert the JSON format to the corresponding binary code streams if the property setting scenario is
involved.
   } else if (msgType == MSG_TYPE_PROPERTIES_GET) {
       // Property query format example: {"msg_type":"properties_get","service_id":"Temperature"}
      // Convert the JSON format to the corresponding binary code streams if the property query scenario is
involved.
   } else if (msgType == MSG_TYPE_MESSAGE_DOWN) {
      // Message delivery format example: {"msg_type":"messages","content":"hello"}
      // Convert the JSON format to the corresponding binary code streams if the message delivery scenario
is involved.
   }
  // Return the encoded binary data.
   return payload;
}
// Parse the message type based on the topic name.
function topicParse(topic) {
   for (var type in TOPIC_REG_EXP) {
       var pattern = TOPIC_REG_EXP[type];
```



```
        if (pattern.test(topic)) {
            return type;
        }
    }
    return '';
}
// Convert an 8-bit unsigned integer into a byte array.
function buffer_uint8(value) {
    var uint8Array = new Uint8Array(1);
    var dataView = new DataView(uint8Array.buffer);
    dataView.setUint8(0, value);
    return [].slice.call(uint8Array);
}

// Convert a 16-bit unsigned integer into a byte array.
function buffer_int16(value) {
    var uint8Array = new Uint8Array(2);
    var dataView = new DataView(uint8Array.buffer);
    dataView.setInt16(0, value);
    return [].slice.call(uint8Array);
}
// Convert a 32-bit unsigned integer into a byte array.
function buffer_int32(value) {
    var uint8Array = new Uint8Array(4);
    var dataView = new DataView(uint8Array.buffer);
    dataView.setInt32(0, value);
    return [].slice.call(uint8Array);
}
```

JavaScript Codec Example for NB-IoT Device Access

The following is an example of the JavaScript codec for NB-IoT devices. Developers can develop codecs for data reporting and command delivery of NB-IoT devices based on the example.

```
// Upstream message types
var MSG_TYPE_PROPERTIES_REPORT = 'properties_report'; // Device property reporting
var MSG_TYPE_COMMAND_RSP = 'command_response'; // Command response
//Downstream message type
var MSG_TYPE_COMMANDS = 'commands'; // Command delivery
var MSG_TYPE_PROPERTIES_REPORT_REPLY = 'properties_report_reply'; // Property reporting response
// Message types
var MSG_TYPE_LIST = {
    0: MSG_TYPE_PROPERTIES_REPORT,         // In the code stream, 0 indicates device property reporting.
    1: MSG_TYPE_PROPERTIES_REPORT_REPLY, // In the code stream, 1 indicates a property reporting
response.
    2: MSG_TYPE_COMMANDS,                  // In the code stream, 2 indicates platform command delivery.
    3: MSG_TYPE_COMMAND_RSP                 // In the code stream, 3 indicates a command response from
the device.
};
/*
Example: When a smoke detector reports properties and returns a command response, it uses binary code
streams. The JavaScript script will decode the binary code streams into JSON data that complies with the
product model definition.
Input parameters:
  payload:[0x00, 0x00, 0x50, 0x00, 0x5a]
Output:
  {"msg_type":"properties_report","services":[{"service_id":"smokerdector","properties":
{"level":80,"temperature":90}}]}
Input parameters:
  payload: [0x03, 0x01, 0x00, 0x00, 0x01]
Output:
  {"msg_type":"command_response","request_id":1,"result_code":0,"paras":{"value":"1"}}
*/
function decode(payload, topic) {
    var jsonObj = {};
    // Perform the AND operation on the payload by using 0xFF to obtain the corresponding complementary
code.
```



```
    var uint8Array = new Uint8Array(payload.length);
    for (var i = 0; i < payload.length; i++) {
        uint8Array[i] = payload[i] & 0xff;
    }
    var dataView = new DataView(uint8Array.buffer, 0);
    // Obtain the message type from the first byte of the message code stream.
    var messageId = dataView.getInt8(0);
    // Convert binary data into the format used for property reporting.
    if (MSG_TYPE_LIST[messageId] == MSG_TYPE_PROPERTIES_REPORT) {
        // Set the value of serviceId, which corresponds to smokerdector in the product model.
        var serviceId = 'smokerdector';
        // Obtain the level value from the code stream.
        var level = dataView.getInt16(1);
        // Obtain the temperature value from the code stream.
        var temperature  = dataView.getInt16(3);
        // Convert the data to the JSON format used by property reporting.
        jsonObj = {"msg_type":"properties_report","services":[{"service_id":serviceId,"properties":
{"level":level,"temperature":temperature}}]};
    }else if (MSG_TYPE_LIST[messageId] == MSG_TYPE_COMMAND_RSP) { // Convert binary data to the
format used by a command response.
        var requestId = dataView.getInt8(1);
        var result_code  = dataView.getInt16(2); // Obtain the command execution result from the binary code
stream.
        var value = dataView.getInt8(4); // Obtain the returned value of the command execution result from
the binary code stream.
       // Convert data to the JSON format used by the command response.
        jsonObj = {"msg_type":"command_response","request_id":requestId,"result_code":result_code,"paras":
{"value":value}};
    }
    // Convert data into a string in JSON format.
    return JSON.stringify(jsonObj);
}
/*
Sample data: When a command is delivered, data in JSON format on IoTDA is encoded into a binary code
stream using the encode method of JavaScript.
Input parameters ->

{"msg_type":"commands","request_id":1,"command_name":"SET_ALARM","service_id":"smokerdector","paras
":{"value":1}}
Output ->
    [0x02, 0x00, 0x00, 0x00, 0x01]
Sample data: When a response is returned for property reporting, data in JSON format on the platform is
encoded into a binary code stream using the encode method of JavaScript.
Input parameters ->
    {"msg_type":"properties_report_reply","request":"000050005a","result_code":0}
Output ->
    [0x01, 0x00]
*/
function encode(json) {
    // Convert data to a JSON object.
    var jsonObj = JSON.parse(json);
   // Obtain the message type.
    var msgType = jsonObj.msg_type;
    var payload = [];
   // Convert data in JSON format to binary data.
    if (msgType == MSG_TYPE_COMMANDS) { // Command delivery
        payload = payload.concat(buffer_uint8(2)); // Command delivery
        payload = payload.concat(buffer_uint8(jsonObj.request_id)); // Command ID
        if (jsonObj.command_name == 'SET_ALARM') {
            payload = payload.concat(buffer_uint8(0)); // Command name
        }
        var paras_value = jsonObj.paras.value;
        payload = payload.concat(buffer_int16(paras_value)); // Set the command property value.
    } else if (msgType == MSG_TYPE_PROPERTIES_REPORT_REPLY) { // Response for device property reporting
        payload = payload.concat(buffer_uint8(1)); // Response to property reporting
        if (0 == jsonObj.result_code) {
            payload = payload.concat(buffer_uint8(0)); // The property reporting message is successfully
processed.
        }
```



```
    }
   // Return the encoded binary data.
    return payload;
}
// Convert an 8-bit unsigned integer into a byte array.
function buffer_uint8(value) {
    var uint8Array = new Uint8Array(1);
    var dataView = new DataView(uint8Array.buffer);
    dataView.setUint8(0, value);
    return [].slice.call(uint8Array);
}
// Convert a 16-bit unsigned integer into a byte array.
function buffer_int16(value) {
    var uint8Array = new Uint8Array(2);
    var dataView = new DataView(uint8Array.buffer);
    dataView.setInt16(0, value);
    return [].slice.call(uint8Array);
}
// Convert a 32-bit unsigned integer into a byte array.
function buffer_int32(value) {
    var uint8Array = new Uint8Array(4);
    var dataView = new DataView(uint8Array.buffer);
    dataView.setInt32(0, value);
    return [].slice.call(uint8Array);
}
```

Requirements on the JavaScript Codec Format

Data Decoding Format

In the data parsing scenario, when the platform receives data from a device, it sends the binary code stream in the payload to the JavaScript script by using the decode method. The script calls the decode method to decode the data to the JSON format defined in the product model. The platform has the following requirements on the parsed JSON data:

- **Device Reporting Properties**

```
{
    "msg_type": "properties_report",
    "services": [{
        "service_id": "Battery",
        "properties": {
            "batteryLevel": 57
        },
        "event_time": "20151212T121212Z"
    }]
}
```

Parameter	Mandatory	Type	Description
msg_type	Yes	String	Message type. The value is fixed to **properties_report**.
services	Yes	List<ServiceProperty>	List of device services. For details, see the ServiceProperty structure table.

ServiceProperty Structure

Parameter	Mandatory	Type	Description
service_id	Yes	String	Service ID of the device.
properties	Yes	Object	Service properties, which are defined in the product model associated with the device.
event_time	No	String	UTC time when the device collects data. The format is yyyyMMddTHHmmssZ, for example, **20161219T114920Z**. If this parameter is not carried in the reported data or is in incorrect format, the time when the platform receives the data is used.

- **Responding to the Platform for Property Setting**

```
{
  "msg_type": "properties_set_response",
  "request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
  "result_code": 0,
  "result_desc": "success"
}
```

Parameter	Mandatory	Type	Description
msg_type	Yes	String	Message type. The value is fixed to **properties_set_response**.
request_id	No	String	Unique identifier of the request. If this parameter is carried in a message received by a device, the parameter value must be carried in the response sent to the platform. If the decoded message does not contain this field, the value of **request_id** in the topic is used.
result_code	No	Integer	Execution result. **0** indicates success, and other values indicate failure. If this parameter is not carried, the execution is considered successful.
result_desc	No	String	Description of the property setting response.

- **Responding to the Platform for Property Query**

```
{
"msg_type": "properties_get_response",
"request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
    "services": [
        {
            "service_id": "analog",
            "properties": {
```



```
            "PhV_phsA": "1",
            "PhV_phsB": "2"
        },
        "event_time": "20190606T121212Z"
    }
  ]
}
```

Parameter	Mandatory	Type	Description
msg_type	Yes	String	The value is fixed at **properties_get_response**.
request_id	No	String	Unique identifier of the request. If this parameter is carried in a message received by a device, the parameter value must be carried in the response sent to the platform. If the decoded message does not contain this field, the value of **request_id** in the topic is used.
services	Yes	List<ServiceProperty>	List of device services. For details, see the ServiceProperty structure table.

ServiceProperty Structure

Parameter	Mandatory	Type	Description
service_id	Yes	String	Service ID of the device.
properties	Yes	Object	Service properties, which are defined in the product model associated with the device.
event_time	No	String	UTC time when the device collects data. The format is yyyyMMddTHHmmssZ, for example, **20161219T114920Z**. If this parameter is not carried in the reported data or is in incorrect format, the time when the platform receives the data is used.

- **Responding to the Platform for Command Delivery**

```
{
 "msg_type": "command_response",
 "request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
  "result_code": 0,
 "command_name": "ON_OFF",
 "service_id": "WaterMeter",
 "paras": {
   "value": "1"
 }
}
```


Parameter	Mandatory	Type	Description
msg_type	Yes	String	The value is fixed at **command_response**.
request_id	No	String	Unique identifier of the request. If this parameter is carried in a message received by a device, the parameter value must be carried in the response sent to the platform. If the decoded message does not contain this field, the value of **request_id** in the topic is used.
result_code	No	Integer	Execution result. **0** indicates success, and other values indicate failure. If this parameter is not carried, the execution is considered successful.
response_name	No	String	Response name, which is defined in the product model associated with the device.
paras	No	Object	Response parameters, which are defined in the product model associated with the device.

- **Device Reporting Messages**

```
{
 "msg_type": "message_up",
 "content": "hello"
}
```

Parameter	Mandatory	Type	Description
msg_type	Yes	String	The value is fixed at **message_up**.
content	No	String	Message content.

Data Encoding Format

In the data parsing scenario, when the IoT platform delivers a command, it sends the data in JSON format defined by the product model to the JavaScript script using the encode method. If the data is not in JSON format, encoding and decoding may fail. The script calls the encode method to encode the data in JSON format into binary code streams that can be identified by the device. During encoding, the JSON format transferred from the platform to the script is as follows:

- **Delivering a Device Command**

```
{
 "msg_type": "commands",
  "request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
 "command_name": "ON_OFF",
 "service_id": "WaterMeter",
```



```
  "paras": {
    "value": 1
  }
}
```

Parameter	Mandatory	Type	Description
msg_type	Yes	String	The value is fixed at **commands**.
request_id	Yes	String	Unique ID of a request. The ID is delivered to the device through a topic.
service_id	No	String	Service ID of the device.
command_name	No	String	Command name, which is defined in the product model associated with the device.
paras	No	Object	Command execution parameters, which are defined in the product model associated with the device.

- **Platform Setting Device Properties**

```
{
"msg_type": "properties_set",
"request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
    "services": [{
            "service_id": "Temperature",
            "properties": {
                "value": 57
            }
        },
        {
            "service_id": "Battery",
            "properties": {
                "level": 80
            }
        }
    ]
}
```

Parameter	Mandatory	Type	Description
msg_type	Yes	String	The value is fixed at **properties_set**.
request_id	Yes	String	Unique identifier of the request. If this parameter is carried in a message received by a device, the parameter value must be carried in the response sent to the platform.
services	Yes	List<ServiceProperty>	List of device services.

ServiceProperty Structure

Parameter	Mandatory	Type	Description
service_id	Yes	String	Service ID of the device.
properties	Yes	Object	Service properties, which are defined in the product model.

- **Platform Querying Device Properties**

```
{
  "msg_type": "properties_get",
  "request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
  "service_id": "Temperature"
}
```

Parameter	Mandatory	Type	Description
msg_type	Yes	String	The value is fixed at **properties_get**.
request_id	Yes	String	Unique ID of a request. The ID is delivered to the device through a topic.
service_id	No	String	Service ID of the device.

- **Responding to Property Reporting of NB-IoT Device Access**

```
{
 "msg_type": "properties_report_reply",
 "request": "213355656",
 "result_code": 0
}
```

Parameter	Mandatory	Type	Description
msg_type	Yes	String	The value is fixed at **properties_report_reply**.
request	No	String	Base64-encoded string of property reporting.
result_code	No	Integer	Execution result of property reporting.
has_more	No	Boolean	Whether a cache command exists.

- **Delivering Device Messages**

```
{
 "msg_type": "messages",
 "content": "hello"
}
```


Parameter	Mandatory	Type	Description
msg_type	Yes	String	The value is fixed at **messages**.
content	No	String	Content of command delivery.

3.2.4.4 FunctionGraph-based Development

3.2.4.4.1 Overview

Introduction

FunctionGraph can be utilized to convert binary data into JSON data or vice versa. This feature is used when the device has limited capabilities and can only report basic binary data. FunctionGraph supports Node.js, Python, Java, Go, C#, PHP, Cangjie, and custom runtimes, meeting multiple development requirements. You can check run logs and graphical monitoring data in real time, greatly improving development and debugging efficiency.

NOTE

- FunctionGraph hosts and computes event-driven functions in a serverless context while ensuring high reliability, high scalability, and zero maintenance. All you need to do is write your code and set conditions.
- For details about FunctionGraph billing, see **FunctionGraph Billing Overview**. You pay only for what you use and you are not charged when your code is not running.

NOTICE

Check the following guide about data conversion for different protocols:

- **MQTT(S) Codec Example**
- **NB-IoT (CoAP) Codec Example**

Process

Figure 3-77 Use of FunctionGraph

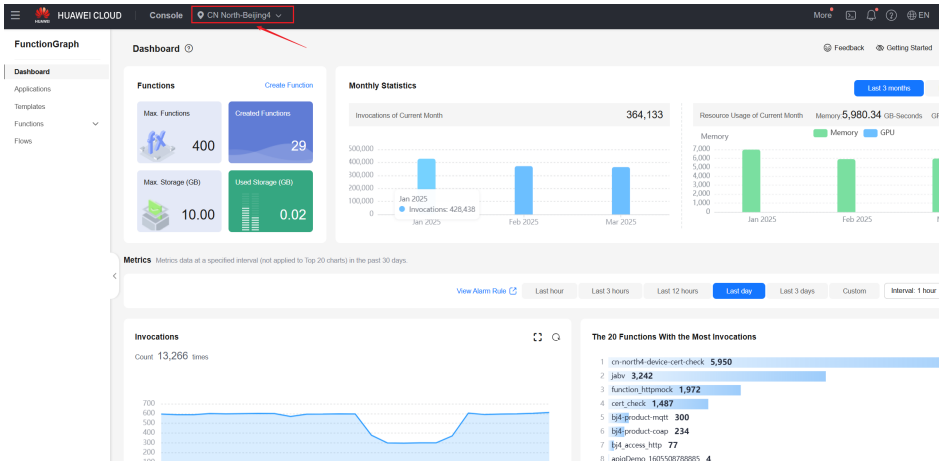
1. Creating a product: Create a CoAP or MQTT product and device on IoTDA. For details, see **Creating a Product**.
   a. Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

b. Choose **Products** in the navigation pane and click **Create Product**. Set the parameters as prompted and click **OK**.

2. Writing the FunctionGraph codec:

a. **Create an event function**. The event function must be created in the same region as that of the created product. Otherwise, the function cannot be referenced by the product. You can check the region in the upper left corner of the console.

Figure 3-78 FunctionGraph-based development - Checking regions



b. Writing codecs. FunctionGraph supports multiple runtime languages, including Python, Node.js, Java, Go, C#, PHP, Cangjie, and custom runtimes. The supported versions vary depending on the languages. For details, see **Supported Programming Languages**.

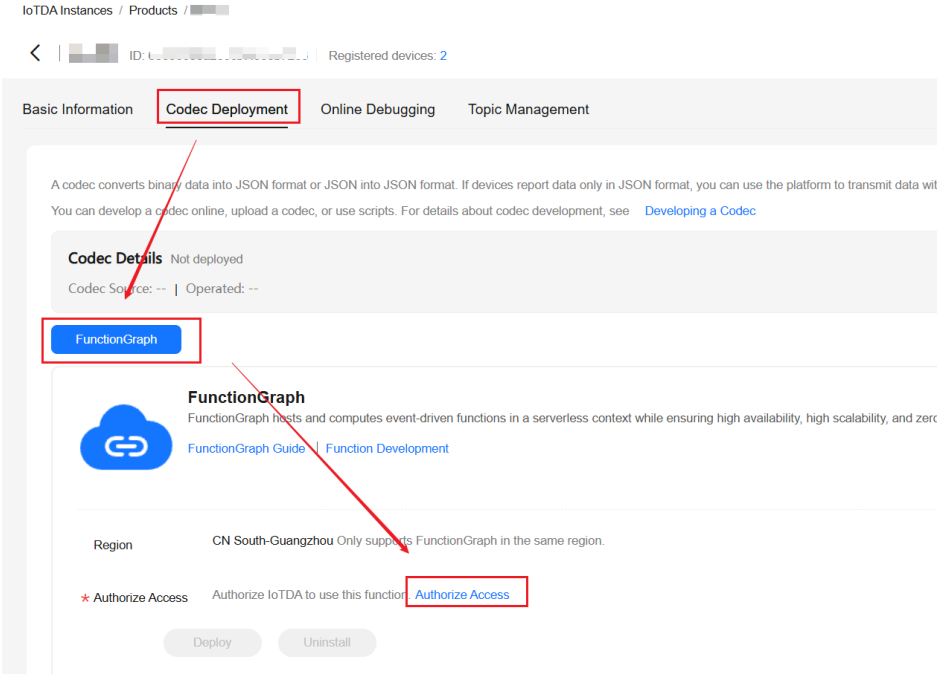
NOTE

Reference: **Creating a Function from Scratch and Executing the Function**.

3. Deploying the FunctionGraph codec:

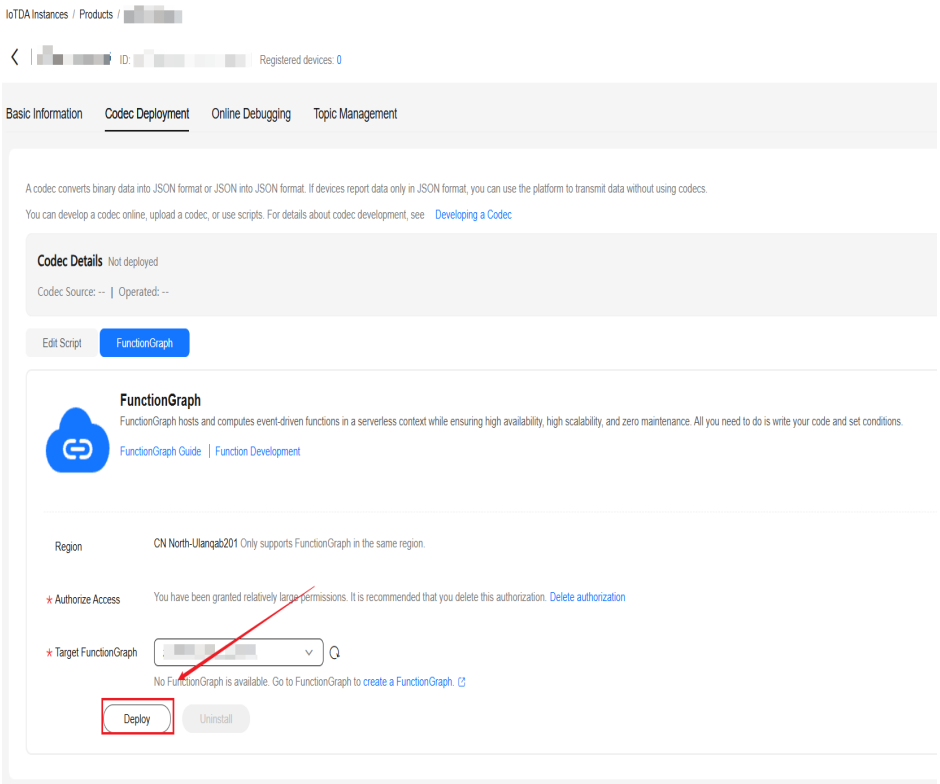
a. Return to the **IoTDA** console, open the product page, click the **Codec Development** tab, and select **FunctionGraph**. If you use the tool for the first time, perform access authorization.

Figure 3-79 FunctionGraph-based Development - Codec authorization



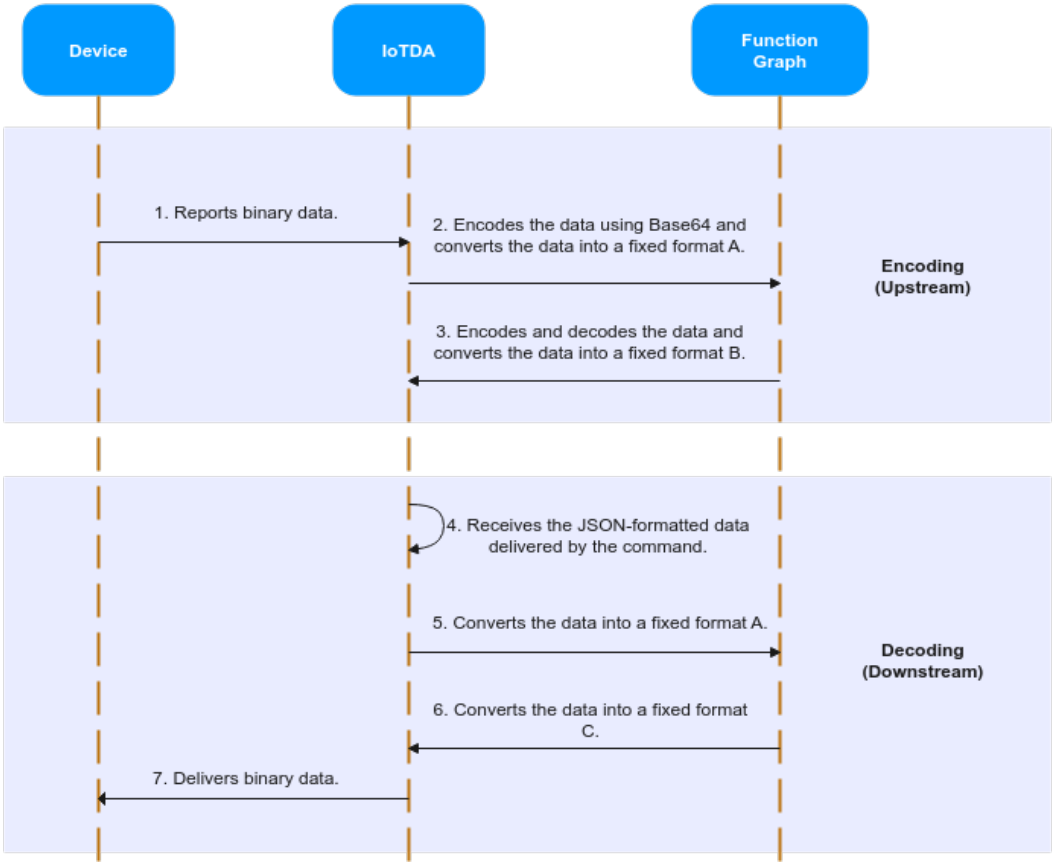
b. After the authorization is successful, select the target function created in 2 and click **Deploy**.

Figure 3-80 FunctionGraph-based Development - Codec deployment



Communications with FunctionGraph Through APIs

Figure 3-81 Process



1. If the reported data is binary data, IoTDA automatically encodes and stores the data using Base64. For example, if a device reports data **[0x01, 0x02]**, the data stored in IoTDA is **AQI=**.

   NOTE

   CoAP products report binary data by default. The data is then encoded using Base64 and sent to the codec. MQTT(S) products report data in the selected data format. The data is then encoded if necessary. For details, see **Creating a Product**.

2. After receiving data, IoTDA transmits the data to FunctionGraph in a specific format if the codec exists. The following table lists the related parameters and data format (fixed format A).

Table 3-3 Upstream data format

Parameter	Mandatory	Type	Description
codecType	Yes	String	**Definition** Execution type. **decode** indicates upstream decoding (from binary code streams to JSON data), and **encode** indicates downstream encoding (from JSON data to binary streams).
message	Yes	String	**Definition** String data in JSON format, which contains the **topic** and **payload** parameters. • **topic**: For MQTT products, the reported topic is carried. For CoAP products, the value is null. • **payload**: Base64 data encoded from the data reported by the device. (For MQTT products, you can select the encoding format on the product page.)

Example of a decoding request sent by IoTDA to FunctionGraph (for CoAP products):

```
{
    "codecType": "decode",
    "message": "{\"topic\": null,\"payload\": \"AABQAFo=\"}"
}
```

Example of a decoding request sent by IoTDA to FunctionGraph (for MQTT products):

```
{
    "codecType": "decode",
    "message": "{\"topic\": \"$oc/devices/661f99d6da14e268414f0af6_longsj123/sys/properties/report
\",\"payload\": \"AABQAFo=\"}"
}
```

3. FunctionGraph decodes the data and returns the result. The following table lists the related parameters and data format (fixed format B).

Table 3-4 Downstream data format

Parameter	Mandatory	Type	Description
status	Yes	String	**Definition** Execution result. **200** indicates success, and other values indicate failure.

Parameter	Mandatory	Type	Description
message	Yes	String	**Definition** String data in JSON format. Used for decoding binary code stream data into JSON data.

Example of a decoding request sent by FunctionGraph to IoTDA:

```
{
    "status": 200,
    "message": "{\"msg_type\":\"properties_report\",\"services\":[{\"service_id\":\"smokerdector
\",\"properties\":{\"level\":258,\"temperature\":3.4}}]}"
 }
```

4. The data initially provided by the platform or applications for delivery is in JSON format and needs to be converted into binary code streams using the codec before final delivery.
5. Before delivering data, IoTDA transmits the data in a specific format to FunctionGraph if the codec exists. The following table lists the related parameters and data format (fixed format A).

Table 3-5 Upstream data format

Parameter	Mandatory	Type	Description
codecType	Yes	String	**Definition** Execution type. **decode** indicates upstream decoding (from binary code streams to JSON data), and **encode** indicates downstream encoding (from JSON data to binary streams).
message	Yes	String	**Definition** String data in JSON format.

Example of an encoding request sent by IoTDA to FunctionGraph:

```
{
   "codecType": "encode",
   "message": "{\"msg_type\":\"commands\",\"service_id\": \"smokerdector\",\"paras\": {\"value\":
1},\"command_name\": \"SET_ALARM\",\"hasMore\": 0,\"request_id\": 1}"
}
```

6. FunctionGraph decodes the data and returns the result. The following table lists the related parameters and data format (fixed format C).

Table 3-6 Downstream data format

Parameter	Mandatory	Type	Description
status	Yes	String	**Definition** Execution result. **200** indicates success, and other values indicate failure.
message	Yes	String	**Definition** String data in JSON format, which contains the **payload** parameter. • **payload**: byte[] data decoded by FunctionGraph.

Example of an encoding request sent by FunctionGraph to IoTDA:

```
{
    "status": 200,
    "message": "{\"payload\":[2,1,0,0,1]}"
}
```

7. The platform delivers the binary code streams encoded by the codec to the device. For example, [2,1,0,0,1].

IoTDA Product Model Data Format

Table 3-7 Data format of a product model

Scenario	Item	Message Type	Supported Protocol	Description
Decoding (from binary code streams to JSON data)	Device reporting properties	properties_report	All	**Device reporting properties**
	Device returning a command response	command_response	All	**Device returning a command response**
	Device returning a response to the platform for property setting	properties_set_response	MQTT/MQTTS	**Device returning a response to the platform for property setting**
	Device returning a response to the platform for property query	properties_get_response	MQTT/MQTTS	**Device returning a response to the platform for property query**

Scenario	Item	Message Type	Supported Protocol	Description
	Device reporting messages	message_up	MQTT/MQTTS	**Device reporting messages**
Encoding (from JSON data to binary code streams)	Platform delivering commands	commands	All	**Platform delivering commands**
	Platform responding to device property reporting	properties_report_reply	NB-IoT (CoAP)	**Platform responding to device property reporting (NB-IoT devices)**
	Platform setting device properties	properties_set	MQTT/MQTTS	**Platform setting device properties**
	Platform querying device properties	properties_get	MQTT/MQTTS	**Platform querying device properties**
	Platform delivering messages	messages	MQTT/MQTTS	**Platform delivering messages**

Data Decoding Format

When the platform receives data from the device, the platform sends the binary code stream in the **payload** to FunctionGraph. FunctionGraph decodes the binary stream into the JSON format defined in the product model. The JSON format can be identified by the platform. The following is the decoded data in JSON format:

```
{
    "status": 200,
    "message": "${Decoded JSON data}"
}
```

${*Decoded JSON data*} is in the JSON format required by the platform.

- Device reporting properties

  ```
  {
     "msg_type": "properties_report",
     "services": [{
        "service_id": "Battery",
        "properties": {
           "batteryLevel": 57
        },
        "event_time": "20151212T121212Z"
     }]
  }
  ```


Table 3-8 Data format of device reporting properties

Parameter	Mandatory	Type	Description
Message Type	Yes	String	Message type. The value is fixed to **properties_report**.
services	Yes	List<ServiceProperty>	List of device services. For details, see the ServiceProperty structure table.

Table 3-9 ServiceProperty structure

Parameter	Mandatory	Type	Description
service_id	Yes	String	Service ID of the device.
properties	Yes	Object	Service properties, which are defined in the product model associated with the device.
event_time	No	String	UTC time when the device collects data. The format is yyyyMMddTHHmmssZ, for example, **20161219T114920Z**. If this parameter is not carried in the reported data or is in incorrect format, the time when the platform receives the data is used.

- **Responding to the Platform for Property Setting**

```
{
  "msg_type": "properties_set_response",
  "request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
  "result_code": 0,
  "result_desc": "success"
}
```

Table 3-10 Data format of device returning a response to the platform for property setting

Parameter	Mandatory	Type	Description
Message Type	Yes	String	Message type. Fixed value: **properties_set_response**

Parameter	Mandatory	Type	Description
request_id	No	String	Unique identifier of the request. If this parameter is carried in a message received by a device, the parameter value must be carried in the response sent to the platform. If the decoded message does not contain this field, the value of **request_id** in the **topic** is used.
result_code	No	Integer	Execution result. **0** indicates success, and other values indicate failure. If this parameter is not carried, the execution is considered successful.
result_desc	No	String	Description of the property setting response.

- **Responding to the Platform for Property Query**

```
{
"msg_type": "properties_get_response",
"request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
   "services": [
      {
         "service_id": "analog",
         "properties": {
            "PhV_phsA": "1",
            "PhV_phsB": "2"
         },
         "event_time": "20190606T121212Z"
      }
   ]
}
```

Table 3-11 Data format of device returning a response to the platform for property query

Parameter	Mandatory	Type	Description
Message Type	Yes	String	The value is fixed at **properties_get_response**.
request_id	No	String	Unique identifier of the request. If this parameter is carried in a message received by a device, the parameter value must be carried in the response sent to the platform. If the decoded message does not contain this field, the value of **request_id** in the topic is used.
services	Yes	List<ServiceProperty>	List of device services. For details, see the ServiceProperty structure table.

Table 3-12 ServiceProperty structure

Parameter	Mandatory	Type	Description
service_id	Yes	String	Service ID of the device.
properties	Yes	Object	Service properties, which are defined in the product model associated with the device.
event_time	No	String	UTC time when the device collects data. The format is yyyyMMddTHHmmssZ, for example, **20161219T114920Z**. If this parameter is not carried in the reported data or is in incorrect format, the time when the platform receives the data is used.

- **Responding to the Platform for Command Delivery**

```
{
  "msg_type": "command_response",
  "request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
   "result_code": 0,
  "command_name": "ON_OFF",
  "service_id": "WaterMeter",
  "paras": {
    "value": "1"
  }
}
```

Table 3-13 Data format of device returning a command response

Parameter	Mandatory	Type	Description
Message Type	Yes	String	The value is fixed at **command_response**.
request_id	No	String	Unique identifier of the request. If this parameter is carried in a message received by a device, the parameter value must be carried in the response sent to the platform. If the decoded message does not contain this field, the value of **request_id** in the topic is used.
result_code	No	Integer	Execution result. **0** indicates success, and other values indicate failure. If this parameter is not carried, the execution is considered successful.
response_name	No	String	Response name, which is defined in the product model associated with the device.

Parameter	Mandatory	Type	Description
paras	No	Object	Response parameters, which are defined in the product model associated with the device.

- **Device Reporting Messages**

```
{
  "msg_type": "message_up",
  "content": "hello"
}
```

Table 3-14 Data format of device reporting messages

Parameter	Mandatory	Type	Description
Message Type	Yes	String	The value is fixed at **message_up**.
content	No	String	Message content.

Data Encoding Format

When the platform delivers data to the device, the platform sends the JSON data defined by the product model to FunctionGraph. (If the data is not in that JSON format, the encoding and decoding may fail.) FunctionGraph encodes the JSON data into binary code streams that can be identified by the device. The following is the data in JSON format sent by the platform to FunctionGraph:

```
{
    "codecType": "encode",
    "message": "${JSON data sent from the platform to FunctionGraph}"
}
```

${*JSON data sent from the platform to FunctionGraph}* is the JSON data sent by the platform to FunctionGraph before encoding.

- Platform delivering commands

```
{
  "msg_type": "commands",
   "request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
  "command_name": "ON_OFF",
  "service_id": "WaterMeter",
  "paras": {
    "value": 1
  }
}
```

Table 3-15 Data format of platform delivering commands

Parameter	Mandatory	Type	Description
Message Type	Yes	String	The value is fixed at **commands**.

Parameter	Mandatory	Type	Description
request_id	Yes	String	Unique ID of a request. The ID is delivered to the device through a topic.
service_id	No	String	Service ID of the device.
command_name	No	String	Command name, which is defined in the product model associated with the device.
paras	No	Object	Command execution parameters, which are defined in the product model associated with the device.

- **Platform Setting Device Properties**

```
{
"msg_type": "properties_set",
"request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
   "services": [{
         "service_id": "Temperature",
         "properties": {
            "value": 57
         }
      },
      {
         "service_id": "Battery",
         "properties": {
            "level": 80
         }
      }
   ]
}
```

Table 3-16 Data format of platform setting device properties

Parameter	Mandatory	Type	Description
Message Type	Yes	String	The value is fixed at **properties_set**.
request_id	Yes	String	Unique identifier of the request. If this parameter is carried in a message received by a device, the parameter value must be carried in the response sent to the platform.
services	Yes	List<ServiceProperty>	List of device services.

ServiceProperty Structure

Table 3-17 ServiceProperty structure

Parameter	Mandatory	Type	Description
service_id	Yes	String	Service ID of the device.
properties	Yes	Object	Service properties, which are defined in the product model.

- **Platform Querying Device Properties**

```
{
  "msg_type": "properties_get",
  "request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
  "service_id": "Temperature"
}
```

Table 3-18 Data format of platform querying device properties

Parameter	Mandatory	Type	Description
Message Type	Yes	String	The value is fixed at **properties_get**.
request_id	Yes	String	Unique ID of a request. The ID is delivered to the device through a topic.
service_id	No	String	Service ID of the device.

- **Platform Responding to Device Property Reporting (NB-IoT Devices)**

```
{
  "msg_type": "properties_report_reply",
  "request": "213355656",
  "result_code": 0
}
```

Table 3-19 Data format of platform responding to device property reporting

Parameter	Mandatory	Type	Description
Message Type	Yes	String	The value is fixed at **properties_report_reply**.
request	No	String	Base64-encoded string of property reporting.
result_code	No	Integer	Execution result of property reporting.
has_more	No	Boolean	Whether a cache command exists.

- **Platform Delivering Messages**

```
{
  "msg_type": "messages",
  "content": "hello"
}
```

Table 3-20 Data format of platform delivering messages

Parameter	Mandatory	Type	Description
Message Type	Yes	String	The value is fixed at **messages**.
content	No	String	Content of command delivery.

3.2.4.4.2 MQTT(S) Codec Example

This section uses a smoke detector as an example to describe how to develop a FunctionGraph codec in JavaScript for reporting properties and delivering commands over MQTT or MQTTS. The codec converts binary data into JSON data and provides a method for debugging.

Defining a Smoke Detector

Scenario

A smoke detector provides the following functions:

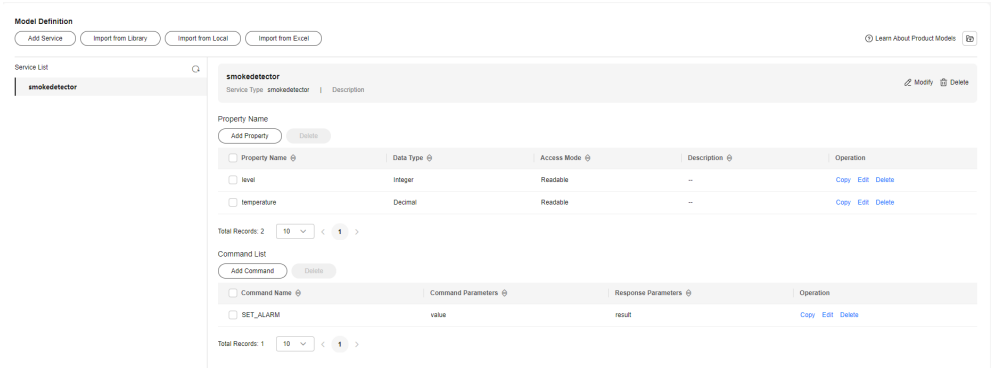
- Reporting smoke alarms (fire severity) and temperature.
- Receiving and running remote control commands, which can be used to enable the alarm function remotely. For example, the smoke detector can report the temperature on the fire scene and remotely trigger a smoke alarm for evacuation.
- The smoke detector has weak capabilities and cannot report data in JSON format defined by the device APIs, but reporting simple binary data.

Product Model

Define the product model on the product details page of the smoke detector.

- **level**: indicates the fire severity.
- **temperature**: indicates the temperature at the fire scene.
- **SET_ALARM**: indicates whether to enable or disable the alarm function. The value **0** indicates that the alarm function is disabled, and **1** indicates that the alarm function is enabled. The response command **result** is used to report the modified alarm value.

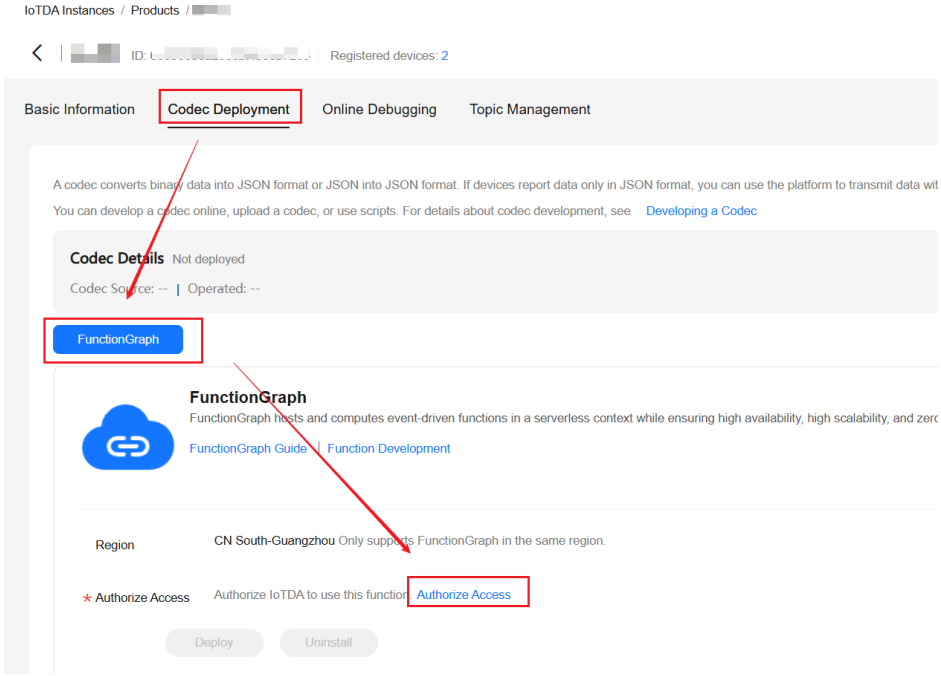
Figure 3-82 Model definition - smokedetector



Developing a Codec

Step 1 On the smoke detector product page, click the **Codec Development** tab, select **FunctionGraph**, and click **Create Function**. If you use the tool for the first time, perform access authorization.

Figure 3-83 FunctionGraph-based Development - Codec authorization



Step 2 On the FunctionGraph console, click **Create Function**. On the displayed page, click **Create from scratch**, enter a function name, and select **Node.js 16.17** as the runtime.

Figure 3-84 Function list - Creating a function

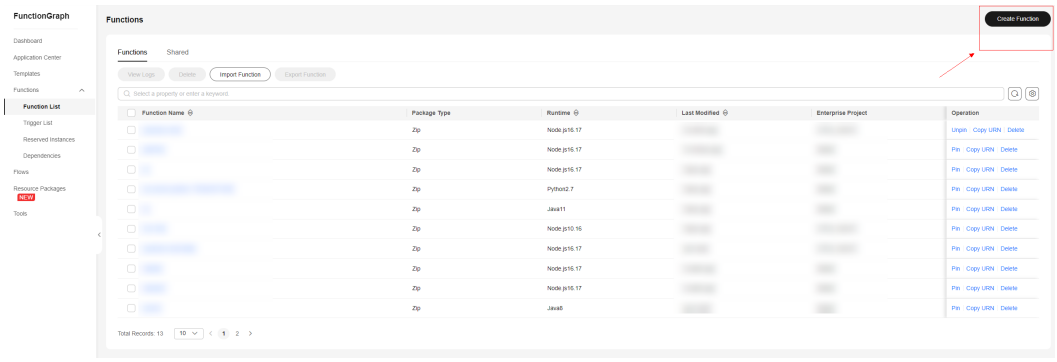
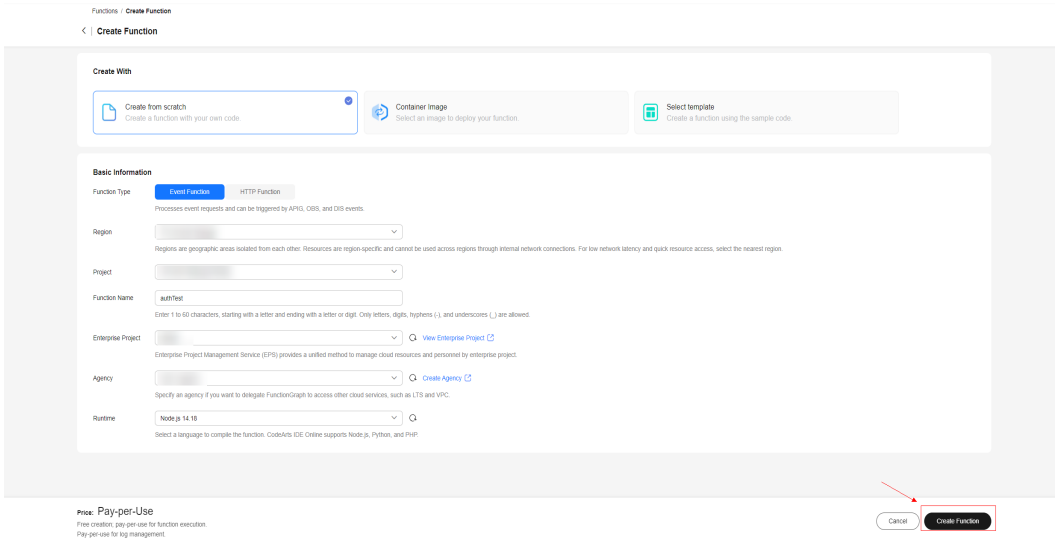


Figure 3-85 Creating a function - Parameters



Step 3 Write a script to convert binary data into JSON data. The script must implement the following methods:

- Decode: Converts the binary data reported by a device into the JSON format defined in the product model. For details about the JSON format requirements, see **Data Decoding Format Definition**.
- Encode: Converts JSON data into binary data supported by the device when the platform sends downstream data to the device. For details about the JSON format requirements, see **Data Encoding Format Definition**.

The following is an example of the JavaScript implemented for the smoke detector. Copy the code to the project and click the button for deploying the code.

```
// Upstream message types
var MSG_TYPE_PROPERTIES_REPORT = 'properties_report'; // Device property reporting
var MSG_TYPE_COMMAND_RSP = 'command_response'; // Command response
var MSG_TYPE_PROPERTIES_SET_RSP = 'properties_set_response'; // Property setting response
var MSG_TYPE_PROPERTIES_GET_RSP = 'properties_get_response'; // Property query response
var MSG_TYPE_MESSAGE_UP = 'message_up'; // Device message reporting
// Downstream message types
var MSG_TYPE_COMMANDS = 'commands'; // Command delivery
var MSG_TYPE_PROPERTIES_SET = 'properties_set'; // Property setting request
var MSG_TYPE_PROPERTIES_GET = 'properties_get'; // Property query request
var MSG_TYPE_MESSAGE_DOWN = 'messages'; // Platform message delivery

// Mapping between topics and upstream MQTT message types
```



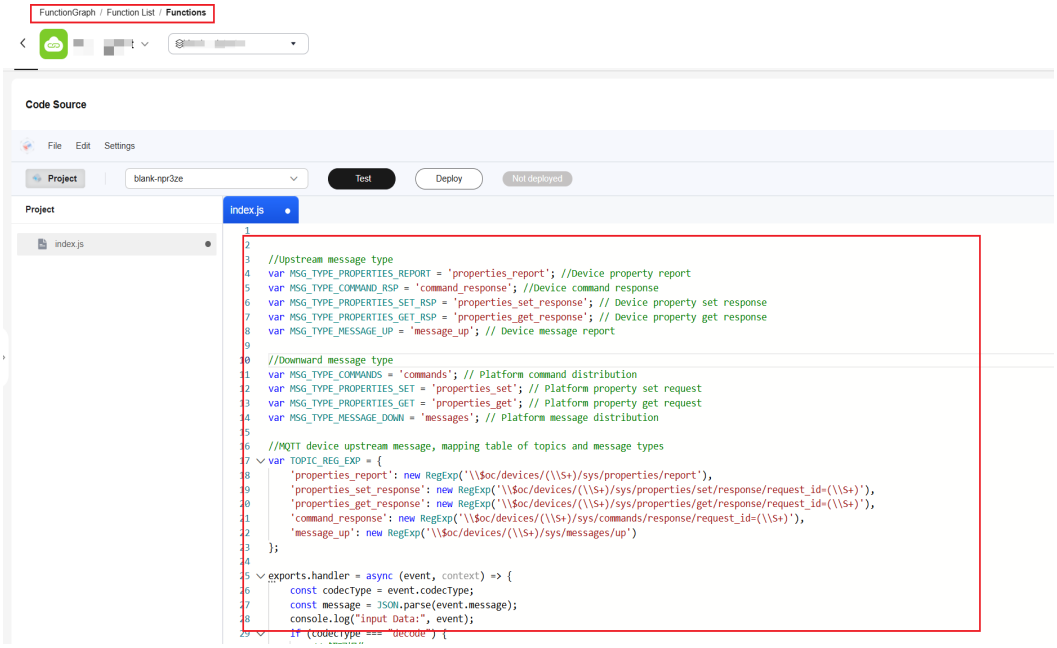
```
var TOPIC_REG_EXP = {
    'properties_report': new RegExp('\\$oc/devices/(\\S+)/sys/properties/report'),
    'properties_set_response': new RegExp('\\$oc/devices/(\\S+)/sys/properties/set/response/request_id=(\\S
+)'),
    'properties_get_response': new RegExp('\\$oc/devices/(\\S+)/sys/properties/get/response/request_id=(\\S
+)'),
    'command_response': new RegExp('\\$oc/devices/(\\S+)/sys/commands/response/request_id=(\\S+)'),
    'message_up': new RegExp('\\$oc/devices/(\\S+)/sys/messages/up')
};
exports.handler = async (event, context) => {
    const codecType = event.codecType;
    const message = JSON.parse(event.message);
    console.log("input Data:", event);
    if (codecType === "decode") {
        // Decoding operation
        return decode(message.payload, message.topic);
    } else if (codecType === "encode") {
        // Encoding operation
        return encode(message);
    }
}
/*
Example: When a smoke detector reports properties and returns a command response, it uses binary code
streams. The JavaScript script will decode the binary code streams into JSON data that complies with the
product model definition.
Input parameters:
  // The first two bytes 0x00 and 0x50 are the value of the level property, and the last two bytes 0x00 and
0x5a are the value of the temperature property.
  payload:[0x00, 0x50, 0x00, 0x5a]
  topic:$oc/devices/cf40f3c4-7152-41c6-a201-a2333122054a/sys/properties/report
Output:
  {"msg_type":"properties_report","services":[{"service_id":"smokerdector","properties":
{"level":80,"temperature":90}}]}
Input parameters:
  // The first byte 0x02 indicates that the command_name is SET_ALARM. The second byte 0x00 indicates
that the command is successfully responded. The last two bytes 0x00 and 0x01 indicate the value of the
command response.
  payload: [0x02, 0x00, 0x00, 0x01]
  topic: $oc/devices/cf40f3c4-7152-41c6-a201-a2333122054a/sys/commands/response/
request_id=bf40f0c4-4022-41c6-a201-c5133122054a
Output:

{"msg_type":"command_response","result_code":0,"command_name":"SET_ALARM","service_id":"smokerdect
or","paras":{"value":"1"}}
*/
// Decoding function
function decode(payload, topic) {
    // Decoding logic
    var binaryString = atob(payload);
    const byteArray = new Uint8Array(binaryString.length);
    for (let i = 0; i < binaryString.length; i++) {
        byteArray[i] = binaryString.charCodeAt(i);
    }
    /*  byteArray is the binary data reported by the device after decoding. You can check whether the
reported data is correctly parsed.*/
    var returnData;
    msgType = topicParse(topic);
        if (msgType == MSG_TYPE_PROPERTIES_REPORT) {
        returnData = decodePropertiesReport(byteArray);
    } else if (msgType == MSG_TYPE_COMMAND_RSP) {
        returnData = decodeCommandRsp(byteArray);
    } else if (msgType == MSG_TYPE_PROPERTIES_SET_RSP) {
        // Convert data to the JSON format used by the property setting response.
        // jsonObj = {"msg_type":"properties_set_response","result_code":0,"result_desc":"success"};
        // returnData = outputData(status, jsonObj)
    } else if (msgType == MSG_TYPE_PROPERTIES_GET_RSP) {
        // Convert data to the JSON format used by the property query response.
        // jsonObj = {"msg_type":"properties_get_response","services":[{"service_id":"analog","properties":
{"PhV_phsA":"1","PhV_phsB":"2"}}]};
```

```
        // returnData = outputData(status, jsonObj)
    } else if (msgType == MSG_TYPE_MESSAGE_UP) {
        // Convert the data to the JSON format used by message reporting.
        // jsonObj = {"msg_type":"message_up","content":"hello"};
        // returnData = outputData(status, jsonObj)
    }
    return returnData;
}
// Encoding function
/*
Sample data: When a command is delivered, data in JSON format on IoTDA is encoded into a binary code stream using the encode method of JavaScript.
Input parameters ->
    {"msg_type":"commands","command_name":"SET_ALARM","service_id":"smokerdector","paras":{"value":1}}
Output ->
    // The first byte 0x01 is used to identify command delivery. The second byte 0x00 indicates command_name = = 'SET_ALARM'. The last two bytes 0x00 and 0x01 are the value of the command properties.
    [0x01, 0x00, 0x00, 0x01]
*/
function encode(data) {
    var msgType = data.msg_type;
    let payload = [];
    var status = 200;
    // Command delivery
    if (msgType == MSG_TYPE_COMMANDS) {
        payload[0] = 0x02; // Command delivery type
        if (data.command_name == 'SET_ALARM') {
            payload[1] = 0x00; // Command name
        }
        // Set the command property value
        payload[2] = (data.paras.value >> 8) & 0xFF;
        payload[3] = data.paras.value & 0xFF;
    } else if (msgType == MSG_TYPE_PROPERTIES_SET) {
        // Response to device property reporting
        // Property setting format example: {"msg_type":"properties_set","services":[{"service_id":"Temperature","properties":{"value":57}}]}
        // Convert the JSON format to the corresponding binary code streams if the property setting scenario is involved.
    } else if (msgType == MSG_TYPE_PROPERTIES_GET) {
        // Property query format example: {"msg_type":"properties_get","service_id":"Temperature"}
        // Convert the JSON format to the corresponding binary code streams if the property query scenario is involved.
    } else if (msgType == MSG_TYPE_MESSAGE_DOWN) {
        // Message delivery format example: {"msg_type":"messages","content":"hello"}
        // Convert the JSON format to the corresponding binary code streams if the message delivery scenario is involved.
    }
    return outputData(status, { "payload": payload });
}
// Parse the message type based on the topic name.
function topicParse(topic) {
    for (var type in TOPIC_REG_EXP) {
        var pattern = TOPIC_REG_EXP[type];
        if (pattern.test(topic)) {
            return type;
        }
    }
    return '';
}
// Property reporting (upstream)
function decodePropertiesReport(byteArray) {
    // Set the value of serviceId, which corresponds to smokerdector in the product model.
    var serviceId = 'smokerdector';
    var level = byteArray[0] * Math.pow(2, 8) + byteArray[1];
    var status = 200;
    var jsonObj;
    if (byteArray.length < 4) {
```



```
        jsonObj = {
            "msg_type": "ERR", "message": "decodePropertiesReport byte length < 5."
        };
        status = 402;
    }
    // Obtain the values of the fourth and fifth values.
    const integerPart = byteArray[2]; // Third value
    const decimalPart = byteArray[3]; // Fourth value
    // Combine the values into decimals.
    const temperature = parseFloat(integerPart + '.' + decimalPart);
    jsonObj = {
        "msg_type": MSG_TYPE_PROPERTIES_REPORT, "services":
            [{ "service_id": serviceId, "properties": { "level": level, "temperature": temperature } }]
    };
    return outputData(status, jsonObj);
}
// Command response (upstream)
function decodeCommandRsp(byteArray) {
    var serviceId = 'smokerdector';
    var command = byteArray[0];
    var command_name = '';
    if (2 == command) {
        command_name = 'SET_ALARM';
    }
    var result_code = byteArray[1]; // Obtain the command execution result from the binary code stream.
    var value = byteArray[2] * Math.pow(2, 8) + byteArray[3]; // Obtain the return value of the command
execution result from the binary code stream.
    // Convert data into the JSON format used by the command response.
    jsonObj = {
        'msg_type': MSG_TYPE_COMMAND_RSP, 'service_id': serviceId, "command_name": command_name,
        'result_code': result_code, 'paras': { 'value': value }
    };
    return outputData(200, jsonObj);
}
// Output the result.
function outputData(status, body) {
    const output =
    {
        'status': status,
        'message': JSON.stringify(body),
    }
    console.log("output Data:", output);
    return output;
}
```


Figure 3-86 FunctionGraph - Copying code to a project



Step 4 Debug the script online. After the script is edited, click **Configure Test Event** on the FunctionGraph console, select a blank template, enter simulated data, and click **Create**. After configuring the test event, click **Test** to obtain the function result and logs.

Simulated data: **payload** is the binary data reported by the device, that is, 0x01, 0x02, 0x03, 0x04. **AQIDBA==** is the result value encoded by the platform using Base64.

```
{
   "codecType": "decode",
   "message": "{\"topic\": \"$oc/devices/device_id/sys/properties/report\",\"payload\": \"AQIDBA==\"}"
}
```

Figure 3-87 FunctionGraph - Adding a test event

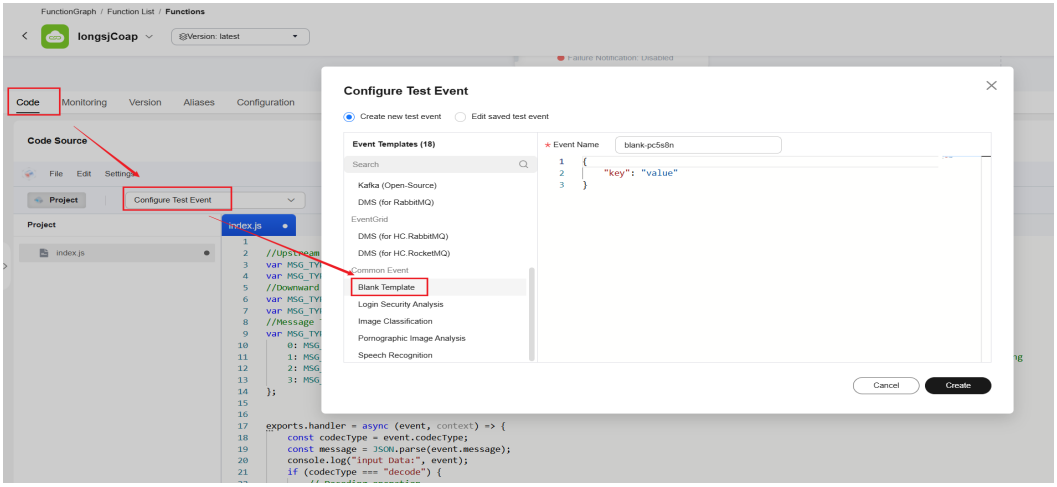
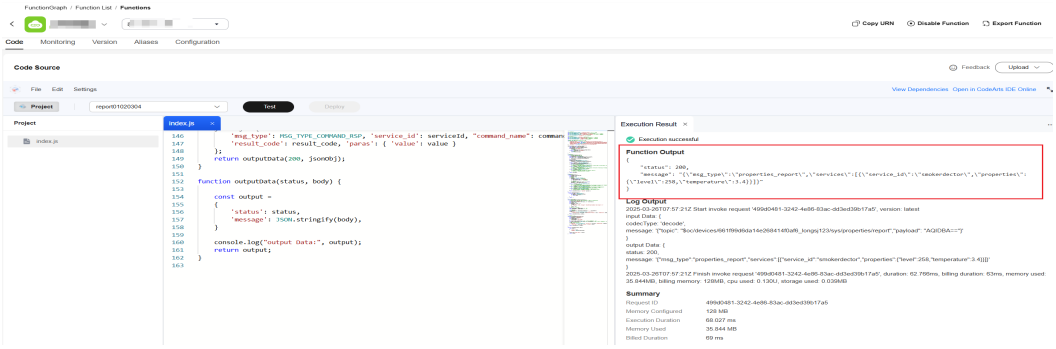
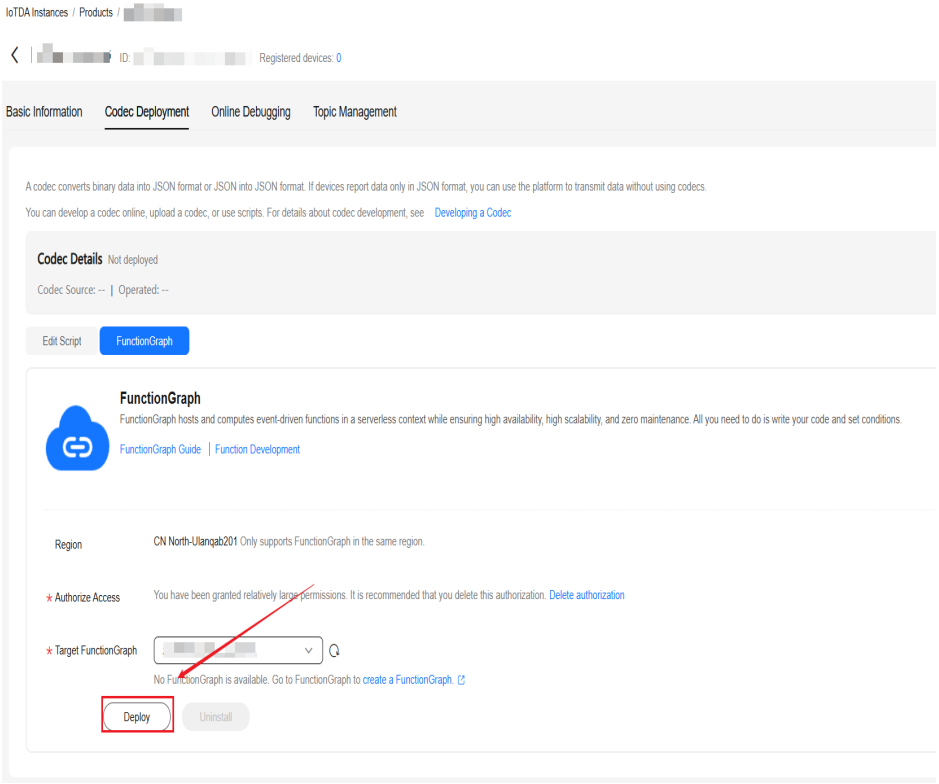


Figure 3-88 FunctionGraph - Test result (MQTT)



Step 5 After the debugging is successful, select the created FunctionGraph function from the drop-down list in **Step 1** and click **Deploy**.

Figure 3-89 FunctionGraph-based Development - Codec deployment



----End

3.2.4.4.3 NB-IoT (CoAP) Codec Example

This section uses a smoke detector as an example to describe how to develop a FunctionGraph codec in JavaScript for reporting properties and delivering commands over CoAP. The codec converts binary data into JSON data and provides a method for debugging.

Defining a Smoke Detector

Scenario

A smoke detector provides the following functions:

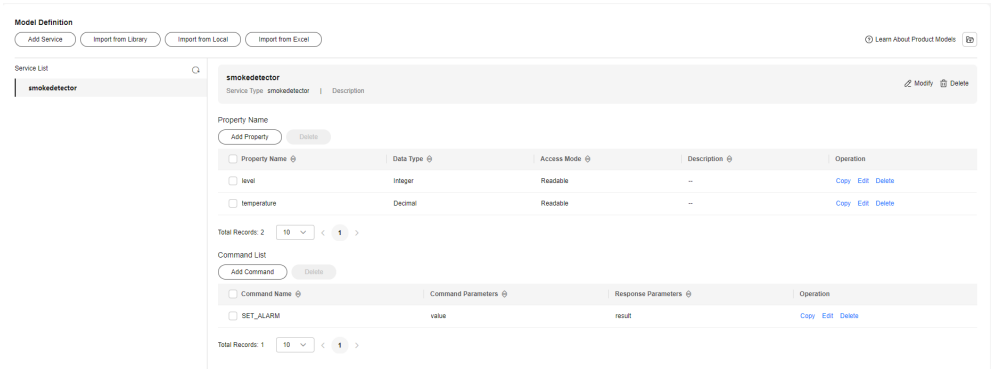
- Reporting smoke alarms (fire severity) and temperature.
- Receiving and running remote control commands, which can be used to enable the alarm function remotely. For example, the smoke detector can report the temperature on the fire scene and remotely trigger a smoke alarm for evacuation.
- The smoke detector has weak capabilities and cannot report data in JSON format defined by the device APIs, but reporting simple binary data.

Product Model

Define the product model on the product details page of the smoke detector.

- **level**: indicates the fire severity.
- **temperature**: indicates the temperature at the fire scene.
- **SET_ALARM**: indicates whether to enable or disable the alarm function. The value **0** indicates that the alarm function is disabled, and **1** indicates that the alarm function is enabled. The response command **result** is used to report the modified alarm value.

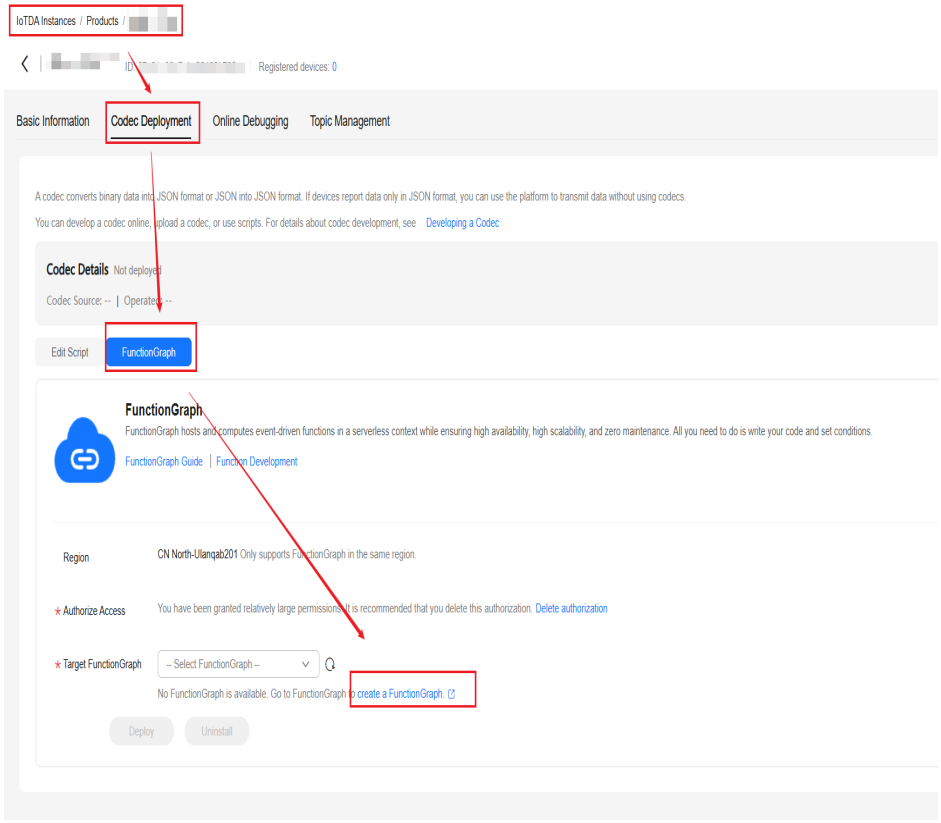
Figure 3-90 Model definition - smokedetector



Developing a Codec

Step 1 On the smoke detector product page, click the **Codec Development** tab, select **FunctionGraph**, and click **Create Function**. If you use the tool for the first time, perform access authorization.

Figure 3-91 FunctionGraph-based Development - Function creation



Step 2 On the FunctionGraph console, click **Create Function**. On the displayed page, click **Create from scratch**, enter a function name, and select **Node.js 16.17** as the runtime.

Figure 3-92 Function list - Creating a function

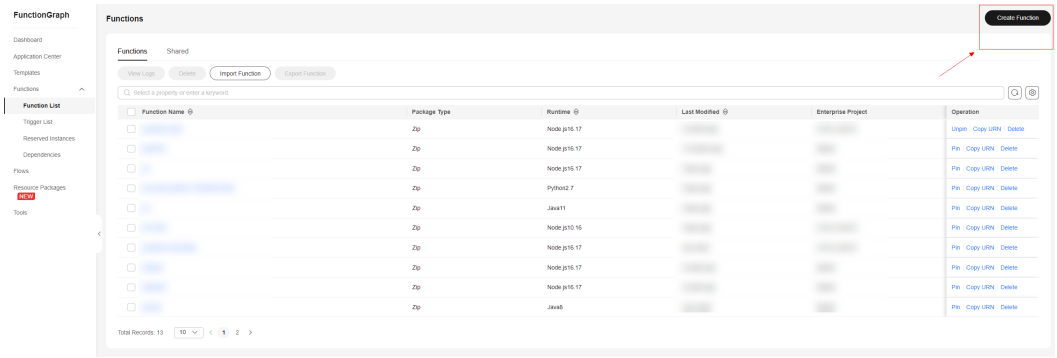
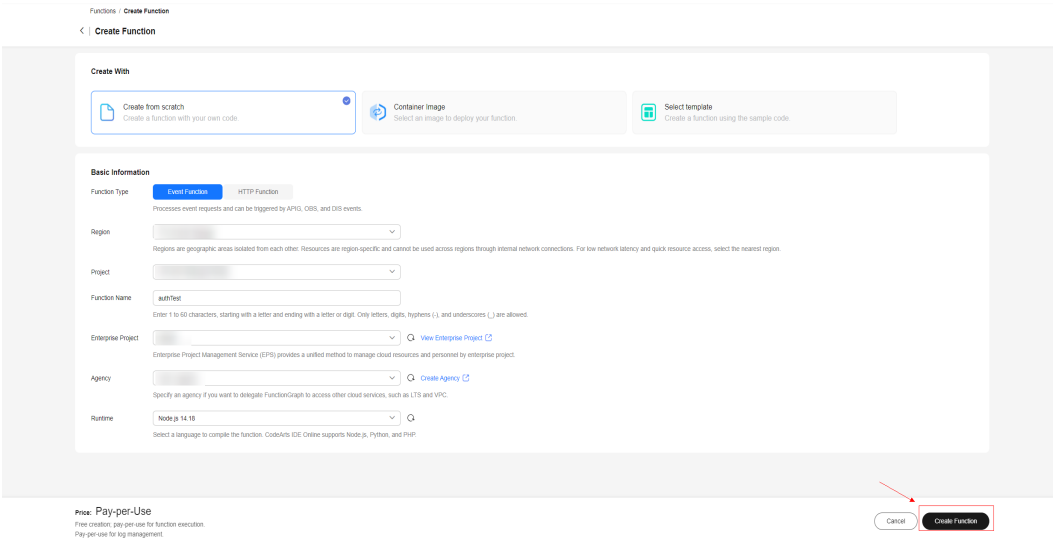


Figure 3-93 Creating a function - Parameters



Step 3 Write a script to convert binary data into JSON data. The script must implement the following methods:

- Decode: Converts the binary data reported by a device into the JSON format defined in the product model. For details about the JSON format requirements, see **Data Decoding Format Definition**.
- Encode: Converts JSON data into binary data supported by the device when the platform sends downstream data to the device. For details about the JSON format requirements, see **Data Encoding Format Definition**.

The following is an example of the JavaScript implemented for the smoke detector. Copy the code to the project.

```
// Upstream message types
var MSG_TYPE_PROPERTIES_REPORT = 'properties_report'; // Device property reporting
var MSG_TYPE_COMMAND_RSP = 'command_response'; // Command response
// Downstream message types
var MSG_TYPE_COMMANDS = 'commands'; // Command delivery
var MSG_TYPE_PROPERTIES_REPORT_REPLY = 'properties_report_reply'; // Property reporting response
// Message types
var MSG_TYPE_LIST = {
    0: MSG_TYPE_PROPERTIES_REPORT,         // In the code stream, 0 indicates device property reporting.
    1: MSG_TYPE_PROPERTIES_REPORT_REPLY, // In the code stream, 1 indicates a property reporting response.
    2: MSG_TYPE_COMMANDS,                  // In the code stream, 2 indicates platform command delivery.
    3: MSG_TYPE_COMMAND_RSP                // In the code stream, 3 indicates a command response from the device.
};
// FunctionGraph entry function
exports.handler = async (event, context) => {
    const codecType = event.codecType;
    const message = JSON.parse(event.message);
    console.log("input Data:", event);
    if (codecType === "decode") {
        // Decoding operation
        return decode(message.payload);
    } else if (codecType === "encode") {
        // Encoding operation
        return encode(message);
    }
}
/*
Example: When a smoke detector reports properties and returns a command response, it uses binary code streams. The JavaScript script will decode the binary code streams into JSON data that complies with the
```

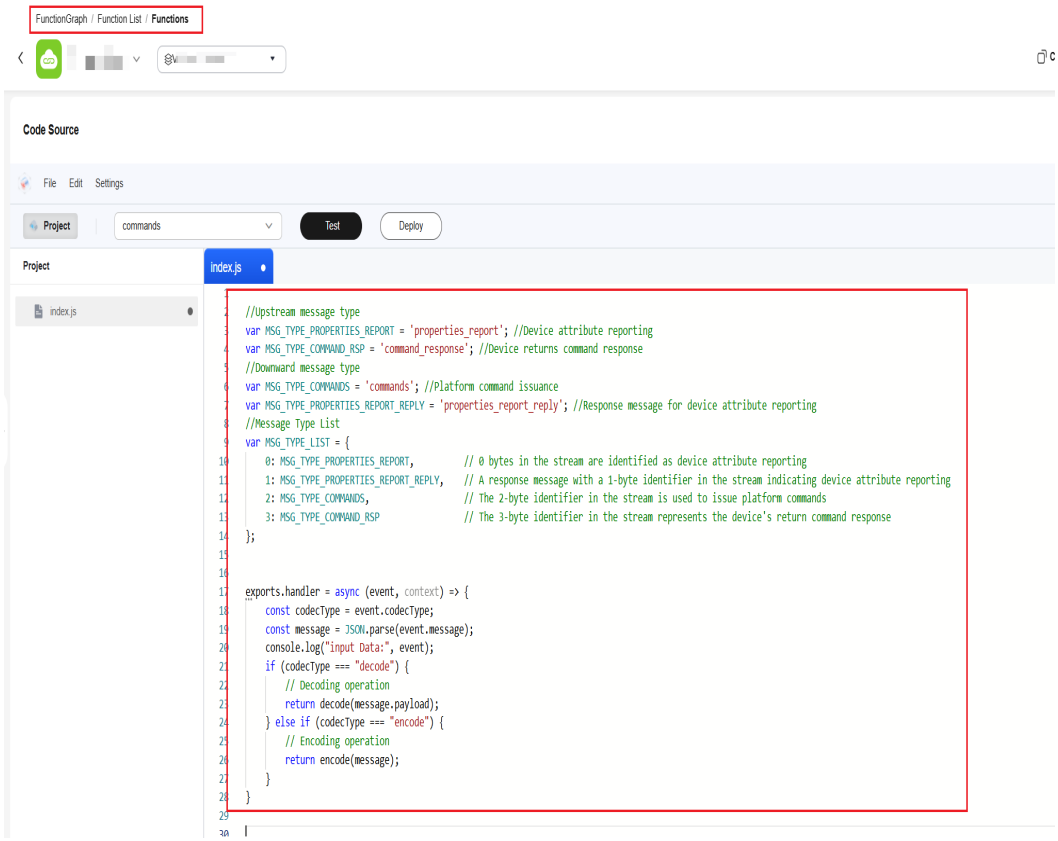


```
product model definition.
Input parameters:
  payload:[0x00, 0x00, 0x50, 0x00, 0x5a]
  payload[0] indicates the data type. 0x00 indicates property reporting. payload[1] and payload[2] indicate the value of the level property. payload[3] and payload[4] indicate the value of the temperature property. payload[3] is the value before the decimal point, and payload[4] is the value after the decimal point.
Output:
  {"msg_type":"properties_report","services":[{"service_id":"smokerdector","properties":{"level":80,"temperature":90}}]}
Input parameters:
  payload: [0x03, 0x01, 0x00, 0x00, 0x01]
  payload[0] indicates the data type. 0x03 indicates that the device returns a command response. payload[1] indicates the value of request_id used to identify the command. payload[2] indicates whether the command is successfully set. If the value is 0, the command is successfully set. payload[3] and payload[4] indicate the values of "value" in the command response.
Output:
  {"msg_type":"command_response","request_id":1,"result_code":0,"paras":{"value":1}}
*/
function decode(payload) {
    // Decoding logic
    var binaryString = atob(payload);
    const byteArray = new Uint8Array(binaryString.length);
    for (let i = 0; i < binaryString.length; i++) {
        byteArray[i] = binaryString.charCodeAt(i);
    }
/* byteArray is the binary data reported by the device after decoding. You can check whether the reported data is correctly parsed.*/
    var returnData;
    var messageId = byteArray[0];
    if (MSG_TYPE_LIST[messageId] == MSG_TYPE_PROPERTIES_REPORT) {
        returnData = decodePropertiesReport(byteArray);
    } else if (MSG_TYPE_LIST[messageId] == MSG_TYPE_COMMAND_RSP) {
        returnData = decodeCommandRsp(byteArray);
    }
    return returnData;
}
/*
Example data:
When a command is delivered, data in JSON format on IoTDA is encoded into a binary code stream using the encode method of JavaScript.
Input parameters ->

{"msg_type":"commands","request_id":1,"command_name":"SET_ALARM","service_id":"smokerdector","paras":{"value":1}}
Output ->
    [0x02, 0x00, 0x00, 0x00, 0x01]
    payload[0] indicates the data type, 0x02 indicates the platform command delivery. payload[1] indicates the command ID. payload[2] indicates the command name (when command_name is SET_ALARM, payload[2] = 0x00). payload[3] and payload[4] indicate the values of the delivered command.
Sample data: When a response is returned for property reporting, data in JSON format on the platform is encoded into a binary code stream using the encode method of JavaScript.
Input parameters ->
    {"msg_type":"properties_report_reply","request":"000050005a","result_code":0}
Output ->
    [0x01, 0x00]
    payload[0] indicates the data type. 0x01 indicates the response message for reporting device properties. payload[1] indicates the device response result. 0x00 indicates success.
*/
function encode(data) {
    var msgType = data.msg_type;
    let payload = [];
    var status = 200;
    // Command delivery
    if (msgType == MSG_TYPE_COMMANDS) {
        payload[0] = 0x02; // Command delivery type
        payload[1] = data.request_id & 0xFF; // Command ID
        if (data.command_name == 'SET_ALARM') {
            payload[2] = 0x00; // Command name
        }
```



```
        // Set the command property value.
        payload[3] = (data.paras.value >> 8) & 0xFF;
        payload[4] = data.paras.value & 0xFF;
    } else if (msgType == MSG_TYPE_PROPERTIES_REPORT_REPLY) {
        // Response to device property reporting
        payload[0] = 0x01; // Response to the device property reporting type
        if (0 == data.result_code) {
            payload[1] = 0x00; // Property reporting processed
        } else {
            payload[1] = 0x01;
            status = 401;
        }
    }
    return outputData(status, { "payload": payload });
}
// Property reporting (upstream)
function decodePropertiesReport(byteArray) {
    // Set the value of serviceId, which corresponds to smokerdector in the product model.
    var serviceId = 'smokerdector';
    var level = byteArray[1] * Math.pow(2, 8) + byteArray[2];
    var status = 200;
    var jsonObj;
    if (byteArray.length < 4) {
        jsonObj = {
            "msg_type": "ERR", "message":"decodePropertiesReport byte length < 5."
        };
        status = 402;
    }
    // Obtain the values of the fourth and fifth values.
    const integerPart = byteArray[3]; // Fourth value
    const decimalPart = byteArray[4]; // Fifth value
    // Combine the values into decimals.
    const temperature = parseFloat(integerPart + '.' + decimalPart);
    jsonObj = {
        "msg_type": MSG_TYPE_PROPERTIES_REPORT, "services":
            [{ "service_id": serviceId, "properties": { "level": level, "temperature": temperature } }]
    };
    return outputData(status, jsonObj);
}
// Command response (upstream)
function decodeCommandRsp(byteArray) {
    var requestId = byteArray[1];
    var result_code = byteArray[2]; // Obtain the command execution result from the binary code stream.
    var value = byteArray[3] * Math.pow(2, 8) + byteArray[4]; // Obtain the return value of the command
execution result from the binary code stream.
    // Convert data into the JSON format used by the command response.
    jsonObj = {
        'msg_type': MSG_TYPE_COMMAND_RSP, 'request_id': requestId,
        'result_code': result_code, 'paras': { 'value': value }
    };
    return outputData(200, jsonObj);
}
function outputData(status, body) {

    const output =
    {
        'status': status,
        'message': JSON.stringify(body),
    }
    console.log("output Data:", output);
    return output;
}
```


Figure 3-94 FunctionGraph - Copying code to a project (CoAP)



Step 4 Debug the script online. After the script is edited, click **Configure Test Event** on the FunctionGraph console, select a blank template, enter simulated data, and click **Create**. After configuring the test event, click **Test** to obtain the function result and logs.

Simulated data: **payload** is the binary data reported by the device, that is, 0x00, 0x00, 0x05, 0x00,0x5a. **AABQAFo=** is the result value encoded by the platform using Base64.

```
{
    "codecType": "decode",
    "message": "{\"topic\": null,\"payload\": \"AABQAFo=\"}"
}
```


Figure 3-95 FunctionGraph - Adding a test event

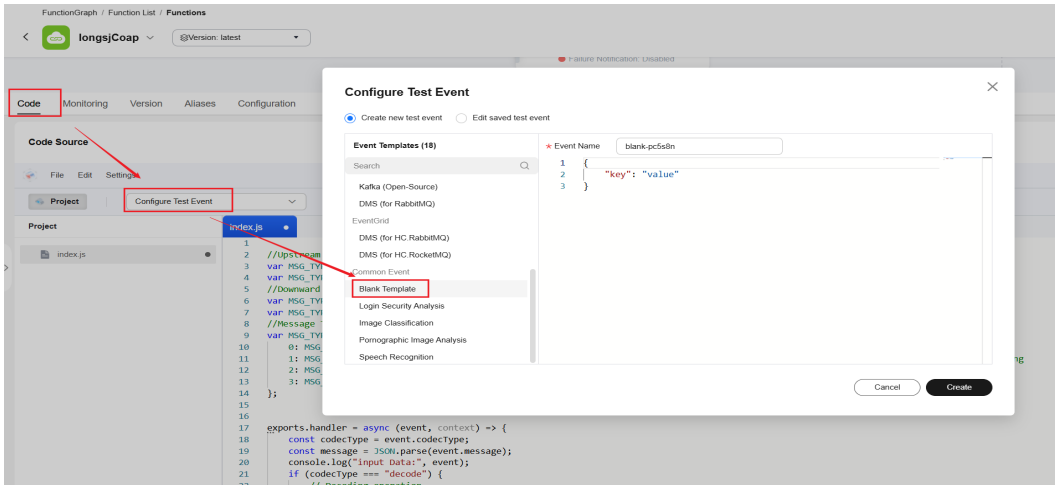
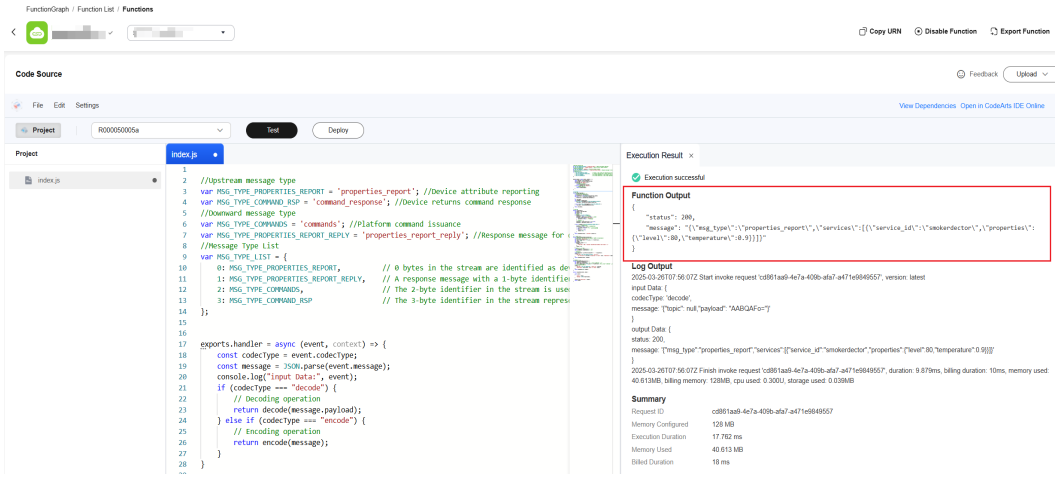
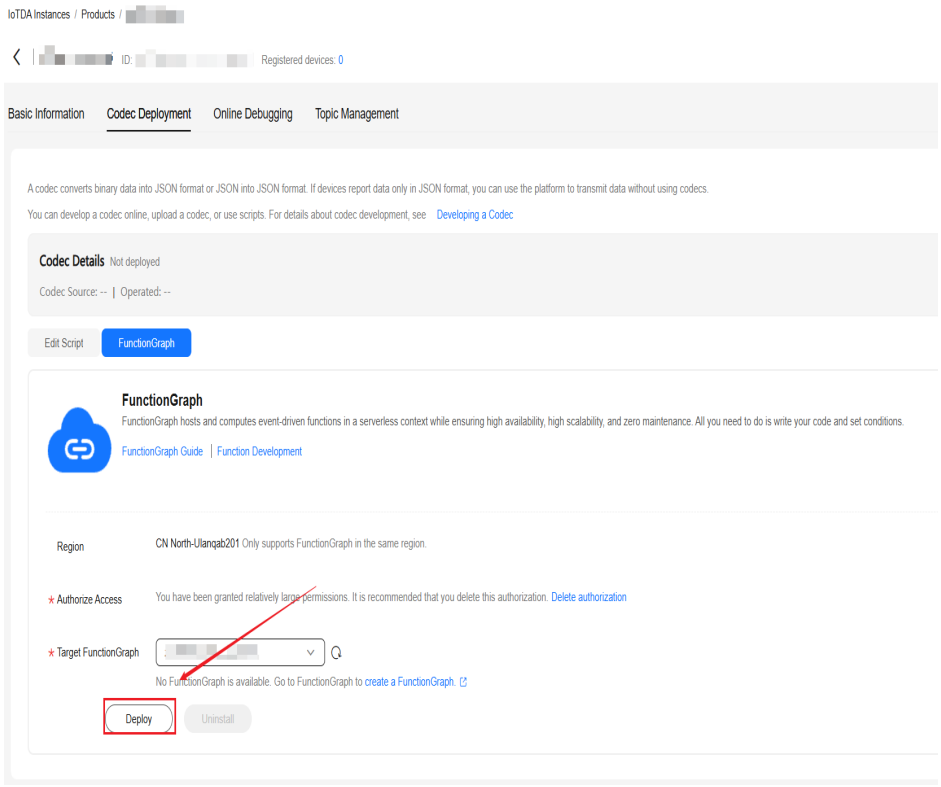


Figure 3-96 FunctionGraph - Test result (CoAP)



Step 5 After the debugging is successful, select the created FunctionGraph function from the drop-down list in **Step 1** and click **Deploy**.

Figure 3-97 FunctionGraph-based Development - Codec deployment



----End

3.2.5 Online Debugging

Overview

After the product model and codec are developed, the application can receive data reported by the device and deliver commands to the device through the platform.

IoTDA provides application and device simulators for you to commission data reporting and command delivery before developing real applications and physical devices. You can also use the application simulator to verify the service flow after the physical device is developed.

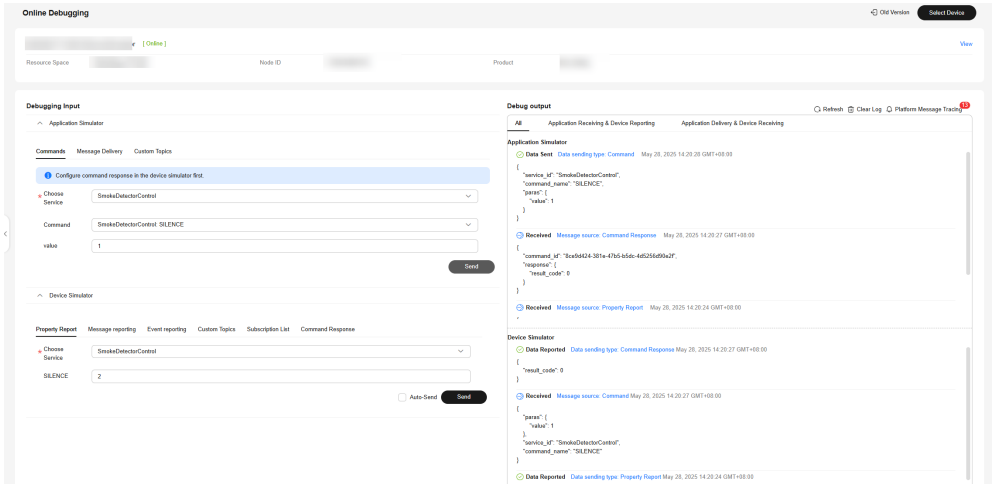
Debugging a Product by Using a Virtual Device

When both device development and application development are not completed, you can create virtual devices and use the application simulator and device simulator to test product models and codecs. The online debugging page consists of the following parts:

1. Device information area (upper part): displays the basic information about the device that is being debugged, including the device name, device status, device ID, resource space, and product.
2. Application simulator area (upper left corner): You can simulate an application to deliver commands, messages, and messages with custom topics.

3. Device simulator area (lower left corner): You can simulate a device to report properties, messages, events, and messages with custom topics, and set command responses.
4. Application simulator record area (upper right corner): displays the data received and delivered by the application.
5. Device simulator record area (lower right corner): displays the data reported and received by the device.

Figure 3-98 Online debugging - Virtual device structure



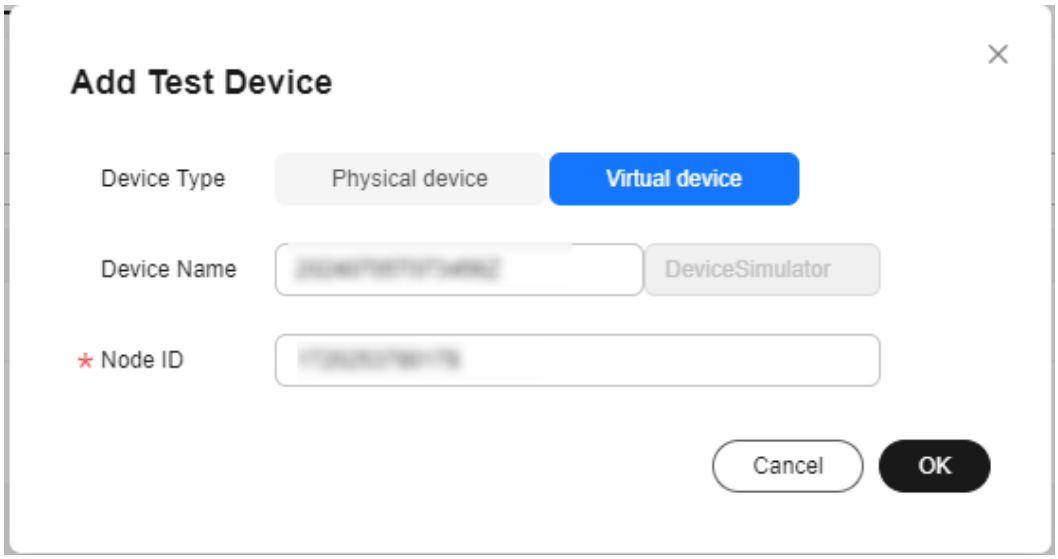
To debug a virtual device online, perform the following steps:

Step 1 On the product details page, click the **Online Debugging** tab and click **Add Test Device**.

Step 2 In the **Add Test Device** dialog box, select **Virtual device** for **Device Type** and click **OK**. The virtual device name contains **DeviceSimulator**. Only one virtual device can be created for each product.

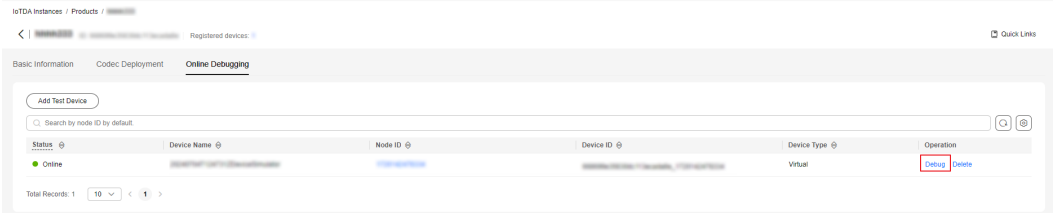
Step 3 In the device list, select the new virtual device.

Figure 3-99 Online debugging - Creating a virtual device



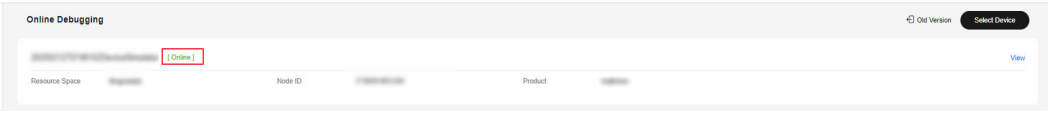
Step 4 Click **Debug** on the right.

Figure 3-100 Entering debugging



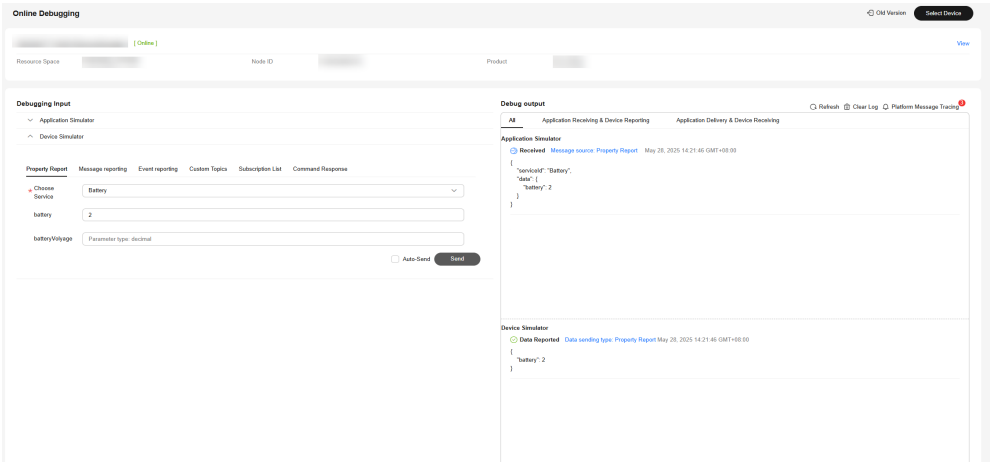
Step 5 On the displayed page, the device status is **Online**.

Figure 3-101 Online Commissioning - Online devices



Step 6 In the **Device Simulator** area, select the usage scenario as required. Options: property reporting, message reporting, event reporting, and data reporting via custom topics. For example, to report a property, click the property reporting tab, select the target service, enter the property value, and click **Send**. Check the reported properties in the device simulator record area on the right. Check the property values received by the application simulator in the application simulator record area.

Figure 3-102 Online debugging - Simulating data reporting (Battery)



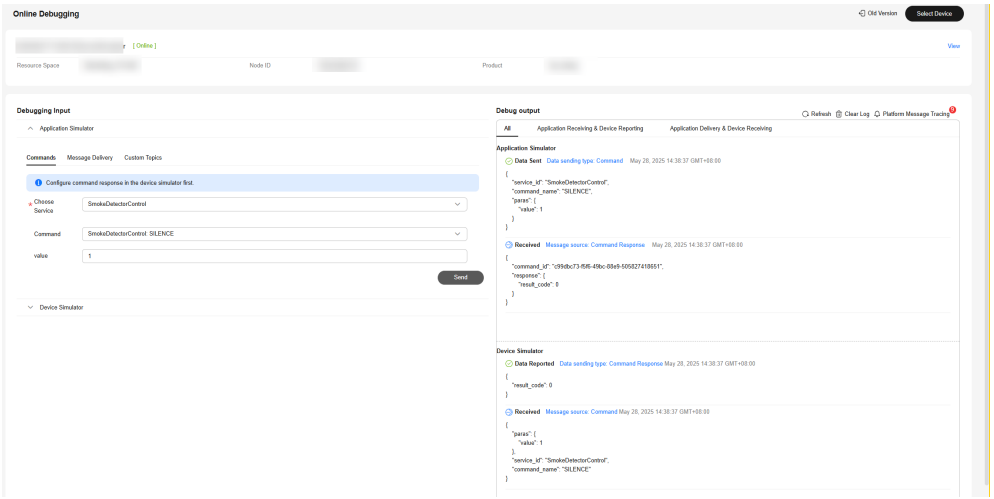
Step 7 In the **Application Simulator** area, select the usage scenario. Options: command delivery, message delivery, and message delivery via custom topics. For example, to deliver a command, click the command delivery tab, select the target service and command, enter the command value, and click **Send**. Check the delivered command and the received command response in the application simulator record area on the right, and check the command received by the device and the command response reported by the device in the device simulator record area.

NOTE

For command delivery, you can set the response reported by the device to the platform on the command response tab page of the device simulator.

For message delivery via custom topics, you can use the device to subscribe to the target topic on the subscription tab page of the device simulator.

Figure 3-103 Online debugging - Command delivery



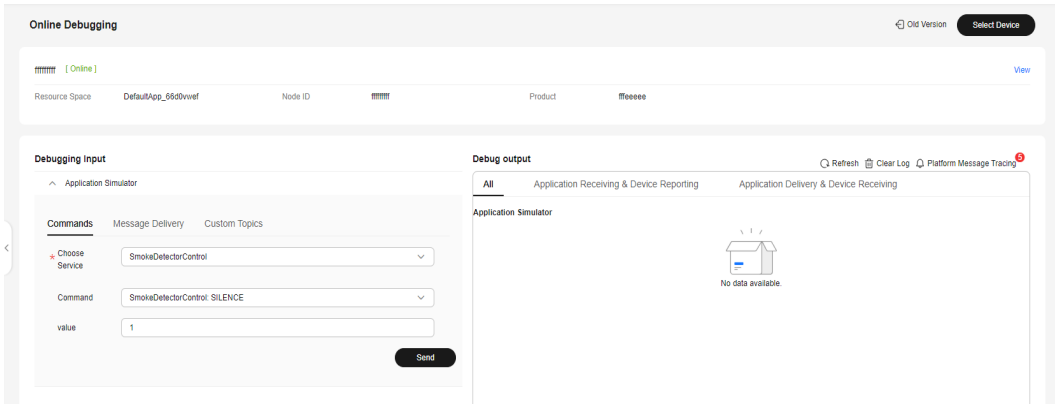
----End

Debugging a Product by Using a Physical Device

When the device development is complete but the application development is not, you can add physical devices and use the application simulator to test devices, product models, and codecs. The physical device debugging page consists of the following parts:

1. Device information area (upper part): displays the basic information about the device that is being debugged, including the device name, device status, device ID, resource space, and product.
2. Application simulator area (left part): You can simulate an application to deliver commands, messages, and messages with custom topics.
3. Application simulator record area: displays the data received and delivered by the application.

Figure 3-104 Online debugging - Physical device structure

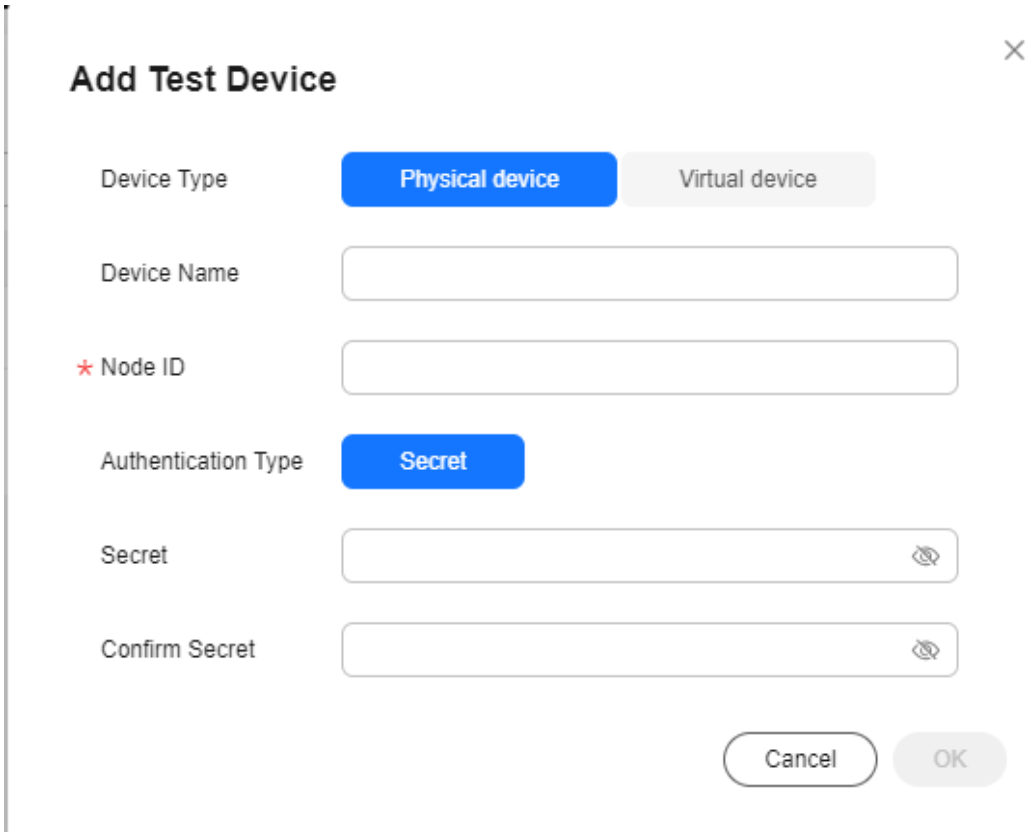


Next, you can create a physical device for online debugging.

Step 1 On the product details page, click the **Online Debugging** tab and click **Add Test Device**.

Step 2 In the **Add Test Device** dialog box, select **Physical device** for **Device Type**, set the parameters of the device, and click **OK**.

Figure 3-105 Online debugging - Adding a test device

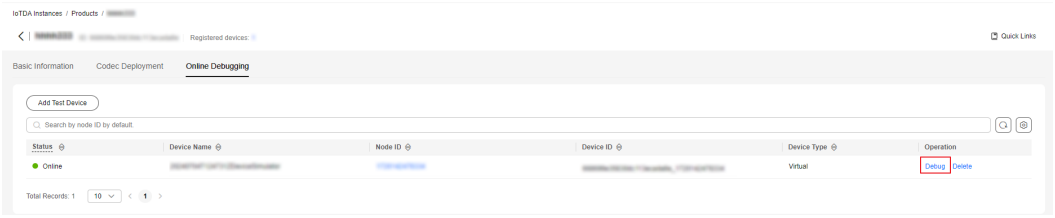


Note: If DTLS is used for access, keep the key secure.

> **NOTE**
>
> The newly added device is in the inactive state. In this case, online debugging cannot be performed. For details, see **Device Connection Authentication**. After the device is connected to the platform, perform the debugging.

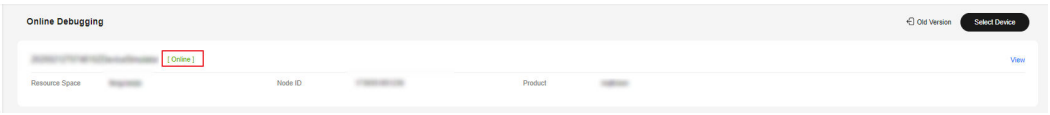
Step 3 Click **Debug** to access the debugging page.

Figure 3-106 Entering debugging



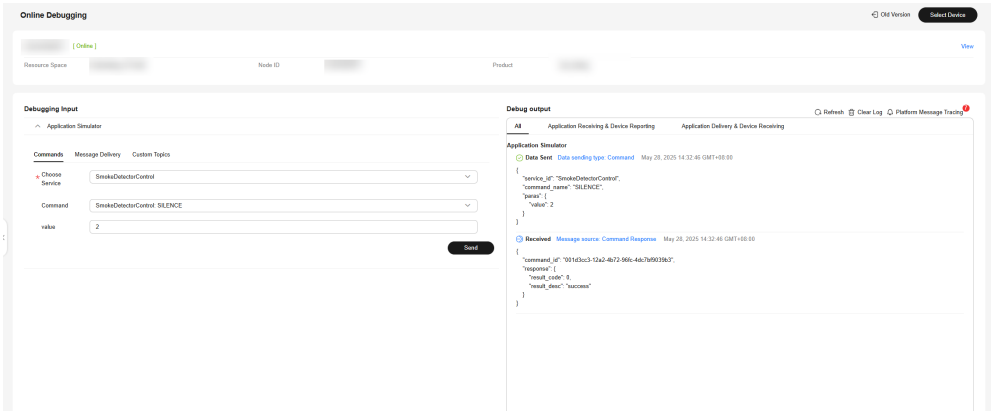
Step 4 On the displayed page, the device status is **Online**.

Figure 3-107 Online Commissioning - Online devices



Step 5 In the **Application Simulator** area, select the usage scenario. Options: command delivery, message delivery, and message delivery via custom topics. For example, to deliver a command, click the command delivery tab, select the target service and command, enter the command value, and click **Send**. Check the delivered command and the received command response in the application simulator record area on the right. Your physical device can receive the delivered commands and perform corresponding actions.

Figure 3-108 Online debugging - Physical devices



----End

3.3 Device Registration

3.3.1 Registering a Device

A device is a physical entity that belongs to a product. Each device has a unique ID. It can be a device directly connected to the platform, or a gateway that connects child devices to the platform. You can register a physical device with the platform, and use the device ID and secret allocated by the platform to connect your SDK-integrated device to the platform.

The platform allows an application to call the API for **creating a device** to register an individual device. Alternatively, you can register an individual device on the IoTDA console. This section describes the procedure on the IoTDA console.

Procedure

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **All Devices**. On the displayed page, click **Register Device**, set parameters based on the table below, and click **OK**.

Figure 3-109 Device - Registering a secret device

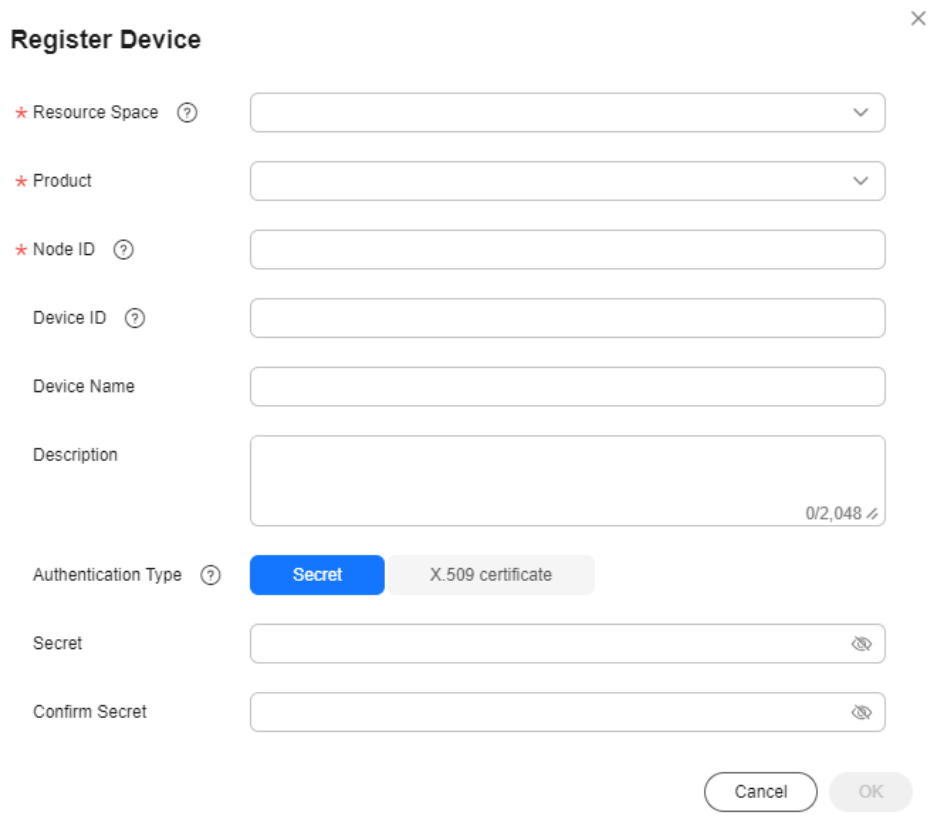
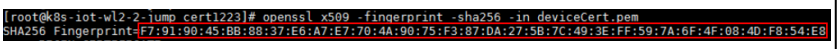
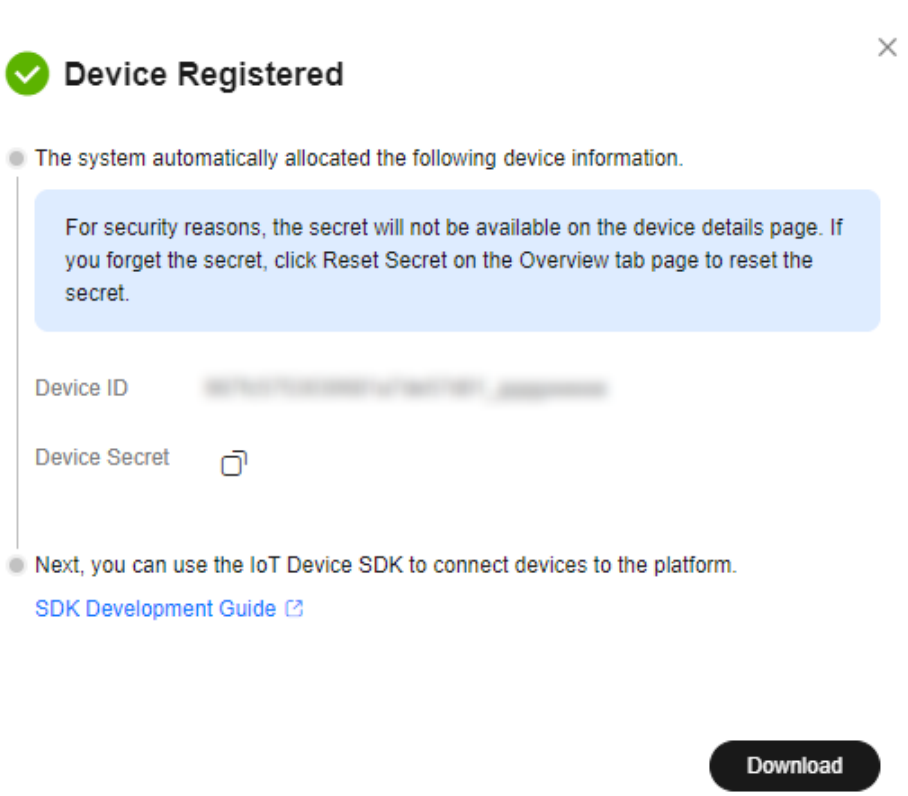


Table 3-21 Registering a device with secret

Parameter	Description
Resource Space	Select the resource space to which a device belongs.
Product	Select the product to which the device belongs. You can select a product only after it is defined. If no product is available, create a product by following the instructions provided in **Creating a Product**.
Node ID	Set this parameter to the IMEI, MAC address, or serial number of the device. If the device is not a physical one, set this parameter to a custom string that contains letters, digits, hyphens (-), and underscores (_).
Device ID	Enter a unique device ID. If this parameter is carried, the platform will use the parameter value as the device ID. Otherwise, the platform will allocate a device ID, which is in the format of *product_id_node_id*.
Device Name	Customize the device name.
Description	Customize device description.
Authentication Type	• **Secret**: The device uses the secret for identity verification. • **X.509 certificate**: The device uses an X.509 certificate for identity verification.
Secret	Customize the secret used for device access. If the secret is left blank, the platform automatically generates one.
Fingerprint	This parameter is displayed when **Authentication Type** is set to **X.509 certificate**. Import the fingerprint corresponding to the **preset device certificate on the device side**. You can run **openssl x509 -fingerprint -sha256 -in deviceCert.pem** in the OpenSSL view to query the fingerprint.  Delete the colons (:) from the obtained fingerprint when filling it.

Save the device ID and secret. They are used for authentication when the device attempts to access the platform.

Figure 3-110 Device - Device registered



NOTE

If the secret is lost, you can **update the secret**. The secret generated during device registration cannot be retrieved.

You can delete a device that is no longer used from the device list. Deleted devices cannot be retrieved. Exercise caution.

----End

APIs

- **Query the Device List**
- **Create a Device**
- **Query a Device**
- **Modify a Device**
- **Delete a Device**
- **Reset a Device Secret**

3.3.2 Registering a Batch of Devices

IoTDA allows an application to call the API for **creating a batch task** to register a batch of devices. Alternatively, you can perform batch registration on the IoTDA console. This section describes the procedure on the IoTDA console.

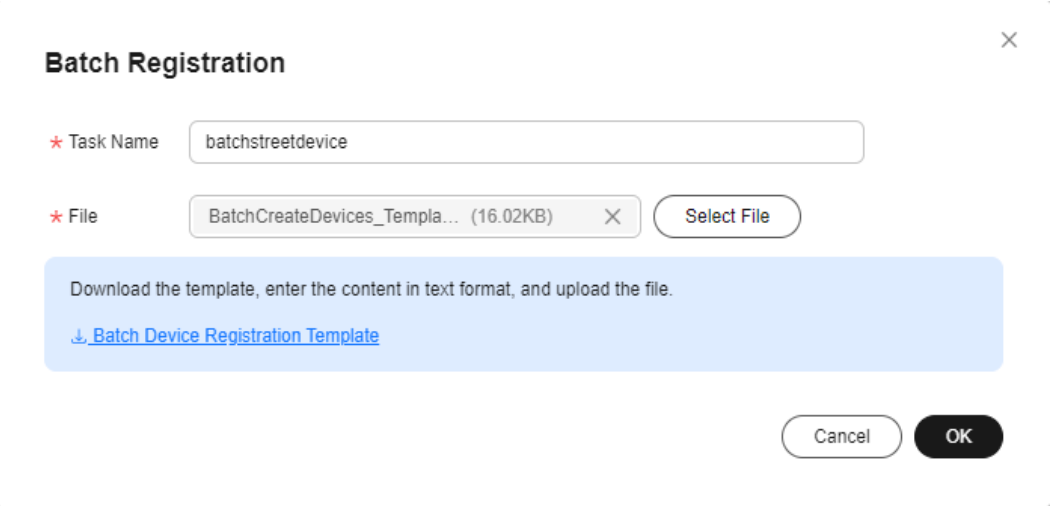
Procedure

Step 1 Access the IoTDA service page and click **Access Console**.

Step 2 In the navigation pane, choose **Devices** > **All Devices**, click the **Batch Registration** tab, and then click **Batch Register**.

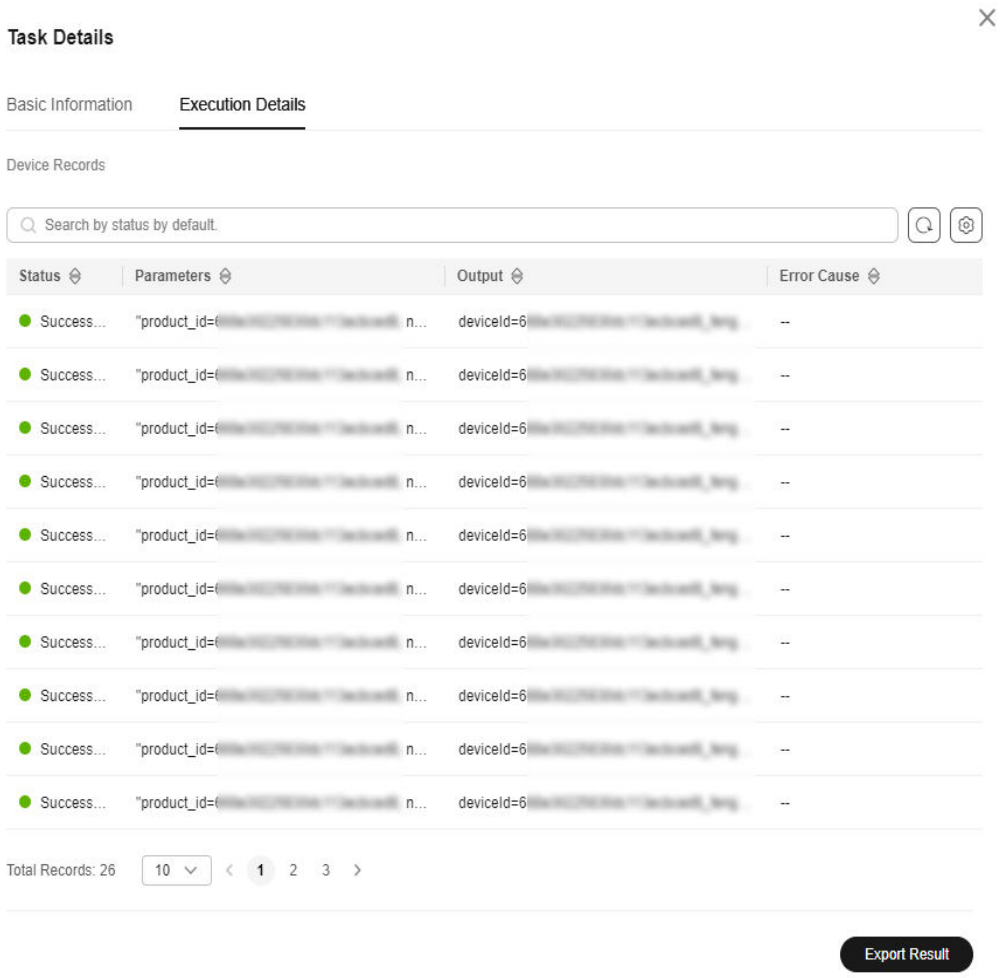
Step 3 In the displayed **Batch Registration** dialog box, enter the task name, download and fill in the **Batch Device Registration Template**, upload the file, and click **OK**.

Figure 3-111 Device - Registering devices in batches



Step 4 If the devices use the native MQTT protocol, click the batch task registration record to open the task execution details, and save the device IDs and secrets generated, which will be used for device access.

Figure 3-112 Batch device registering - Execution details



----End

APIs

- Create a Device
- Query the Batch Task List
- Create a Batch Task
- Query a Batch Task

3.3.3 Registering a Device Authenticated by an X.509 Certificate

An X.509 certificate is a digital certificate used for communication entity authentication. IoTDA allows devices to use their X.509 certificates for authentication. The use of X.509 certificate authentication protects devices from being spoofed.

Before registering a device authenticated by an X.509 certificate, upload the device Certificate Authority (CA) certificate to the platform and bind the device certificate to the device during device registration. This section describes how to upload a

device CA certificate to the platform and register a device that uses the X.509 certificate for authentication.

Constraints

- Only MQTT devices can use X.509 certificates for identity authentication.
- You can upload up to 100 device CA certificates.

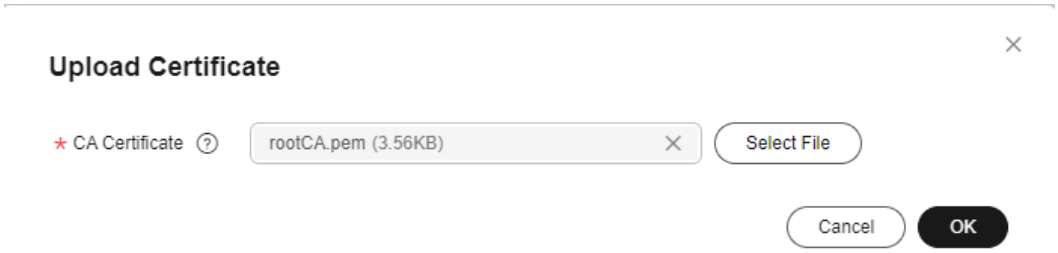
Uploading a Device CA Certificate

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **Device Certificates**. On the **Device CA Certificates** tab page, click **Upload Certificate**.

Step 3 In the displayed dialog box, click **Select File** to add a file, and then click **OK**.

Figure 3-113 Device CA certificate - Uploading a certificate



NOTE

Device CA certificates are provided by device vendors. You can **prepare a commissioning certificate** during commissioning. For security reasons, you are advised to replace the commissioning certificate with a commercial certificate during commercial use. Purchased CA certificates (in formats such as PEM and JKS) can be directly uploaded to the platform.

----End

Creating a Device CA Commissioning Certificate

This section uses the Windows operating system as an example to describe how to use OpenSSL to make a commissioning certificate. The generated certificate is in PEM format.

1. Download and install **OpenSSL**.
2. Open the CLI as user **admin**.
3. Run **cd c:\openssl\bin** (replace **c:\openssl\bin** with the actual OpenSSL installation directory) to access the OpenSSL view.
4. Generate a public/private key pair.
   ```
   openssl genrsa -out rootCA.key 2048
   ```
5. Use the private key in the key pair to generate a CA certificate.
   ```
   openssl req -x509 -new -nodes -key rootCA.key -sha256 -days 1024 -out rootCA.pem
   ```
   Enter the following information as prompted. All parameters can be customized.

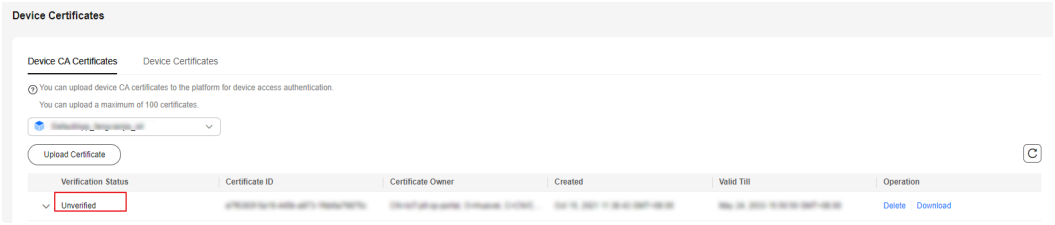
- Country Name (2 letter code) [AU]: country, for example, CN
- State or Province Name (full name) []: state or province, for example, GD
- Locality Name (for example, city) []: city, for example, SZ
- Organization Name (for example, company) []: organization, for example, Huawei
- Organizational Unit Name (for example, section) []: organization unit, for example, IoT
- Common Name (e.g. server FQDN or YOUR name) []: common name, for example, zhangsan
- Email Address []: email address, for example, 1234567@163.com

Obtain the generated CA certificate **rootCA.pem** from the **bin** folder in the OpenSSL installation directory.

Uploading a Verification Certificate

If the uploaded certificate is a commissioning certificate, the certificate status is **Unverified**. In this case, upload a verification certificate to verify that you have the CA certificate.

Figure 3-114 Device CA certificate - Unverified certificate



The verification certificate is created based on the private key of the device CA certificate. Perform the following operations to create a verification certificate:

Step 1 Generate a key pair for the verification certificate.

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

Step 2 Create a certificate signing request (CSR) for the verification certificate.

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

The system prompts you to enter the following information. Set **Common Name** to the verification code and set other parameters as required.

- Country Name (2 letter code) [AU]: country, for example, CN
- State or Province Name (full name) []: state or province, for example, GD
- Locality Name (for example, city) []: city, for example, SZ
- Organization Name (for example, company) []: organization, for example, Huawei
- Organizational Unit Name (for example, section) []: organization unit, for example, IoT
- Common Name (e.g. server FQDN or YOUR name) []: verification code for verifying the certificate. For details on how to obtain the verification code, see **Step 5**.

- Email Address []: email address, for example, 1234567@163.com
- Password[]: password, for example, 1234321
- Optional Company Name[]: company name, for example, Huawei

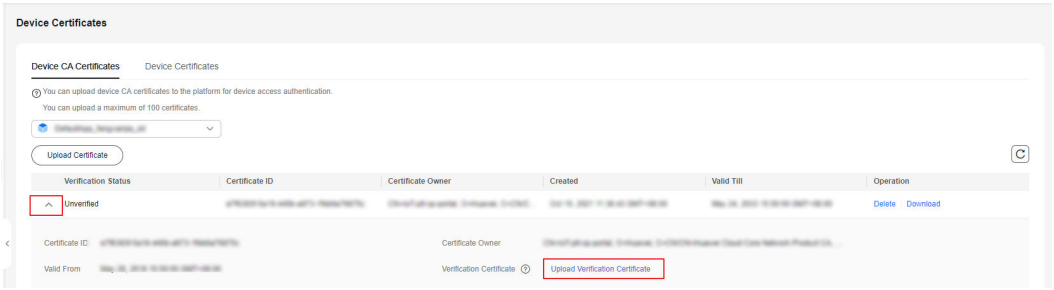
Step 3 Use the CSR to create a verification certificate.

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

Obtain the generated verification certificate **verificationCert.pem** from the **bin** folder of the OpenSSL installation directory.

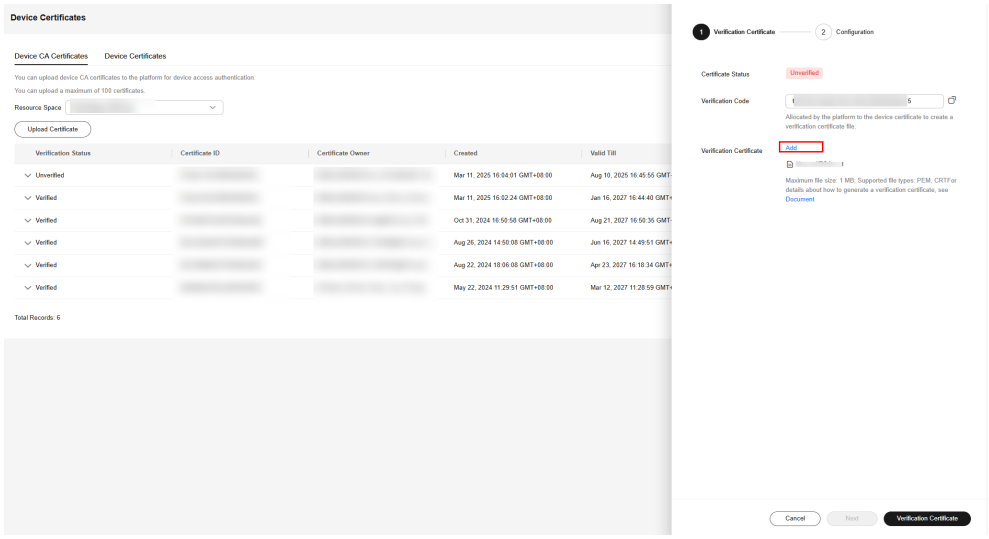
Step 4 Select the corresponding certificate, click , and click **Upload Verification Certificate**.

Figure 3-115 Device CA certificate - Verifying a certificate



Step 5 The verification code is displayed in the dialog box. Click **Select File**, upload the verification certificate, and click **OK**. After the certificate is uploaded, the certificate status changes to **Verified**, indicating that you have the CA certificate.

Figure 3-116 Device CA certificate - Uploading a verified certificate



----End

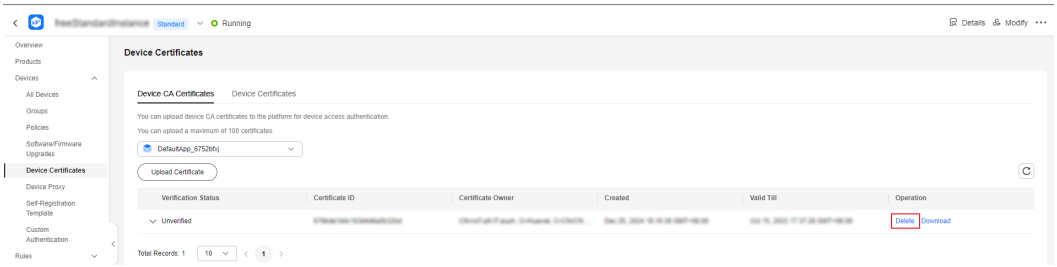
Deleting a Device CA Certificate

You can delete a device CA certificate that is no longer used.

NOTE

Once a service device CA certificate is deleted, devices that rely on it for authentication can no longer access the platform. Back up related data before the deletion.

Figure 3-117 Device CA certificate - Deleting a certificate



Presetting an X.509 Certificate

Before registering an X.509 device, preset the X.509 certificate issued by the CA on the device.

NOTE

The X.509 certificate is issued by the CA. If no commercial certificate issued by the CA is available, you can **create an X.509 commissioning certificate**. Purchased certificates or certificates (in formats such as PEM and JKS) issued by authoritative organizations can be directly uploaded to the platform.

Creating an X.509 Commissioning Certificate

1. Run **cmd** as user **admin** to open the CLI and run **cd c:\openssl\bin** (replace **c:\openssl\bin** with the actual OpenSSL installation directory) to access the OpenSSL view.
2. Generate a public/private key pair.
   ```
   openssl genrsa -out deviceCert.key 2048
   ```
3. Create a CSR for the device certificate.
   ```
   openssl req -new -key deviceCert.key -out deviceCert.csr
   ```
   Enter the following information as prompted. All parameters can be customized.
   - Country Name (2 letter code) [AU]: country, for example, CN
   - State or Province Name (full name) []: state or province, for example, GD
   - Locality Name (for example, city) []: city, for example, SZ
   - Organization Name (for example, company) []: organization, for example, Huawei
   - Organizational Unit Name (for example, section) []: organization unit, for example, IoT
   - Common Name (e.g. server FQDN or YOUR name) []: common name, for example, zhangsan
   - Email Address []: email address, for example, 1234567@163.com
   - Password[]: password, for example, 1234321
   - Optional Company Name[]: company name, for example, Huawei

4. Create a device certificate using CSR.

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

Obtain the generated device certificate **deviceCert.pem** from the **bin** folder in the OpenSSL installation directory.

Registering a Device Authenticated by an X.509 Certificate

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **All Devices**. On the displayed page, click **Register Device**, set parameters based on the table below, and click **OK**.

Figure 3-118 Device - Registering an X.509 device

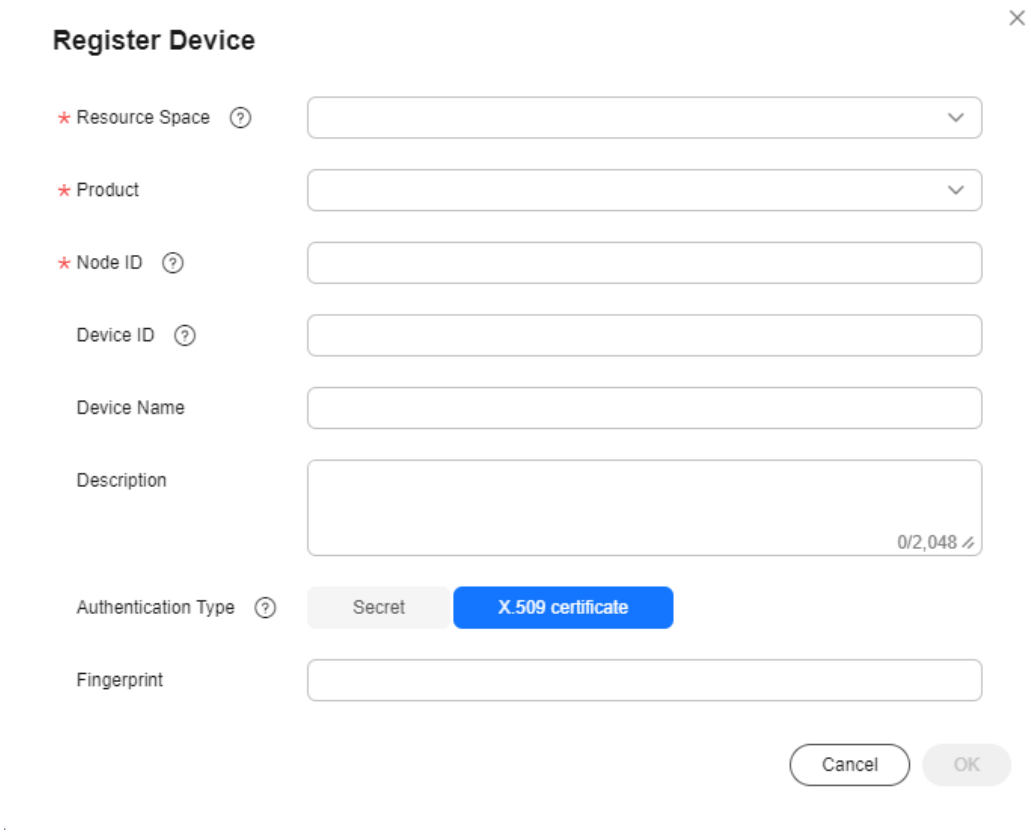
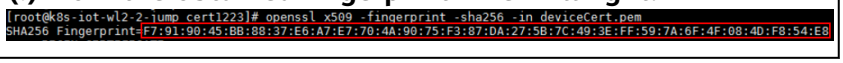


Table 3-22 Registering a device using X.509 certificate

Parameter	Description
Resource Space	Select the resource space to which a device belongs.
Product	Select the product to which the device belongs. Select an existing or **create one**.

Parameter	Description
Node ID	Set this parameter to the IMEI, MAC address, or serial number of the device. If the device is not a physical one, set this parameter to a custom string that contains letters, digits, hyphens (-), and underscores (_).
Device ID	Enter a unique device ID. If this parameter is carried, the platform will use the parameter value as the device ID. Otherwise, the platform will allocate a device ID, which is in the format of *product_id_node_id*.
Device Name	Customize the device name.
Description	Customize device description.
Authentication Type	**X.509 certificate**: The device uses an X.509 certificate for identity verification.
Fingerprint	This parameter is displayed when **Authentication Type** is set to **X.509 certificate**. Import the fingerprint corresponding to the **preset device certificate on the device side**. You can run **openssl x509 -fingerprint -sha256 -in deviceCert.pem** in the OpenSSL view to query the fingerprint. **Note: Delete the colon (:) from the obtained fingerprint when filling it.** 

----End

APIs

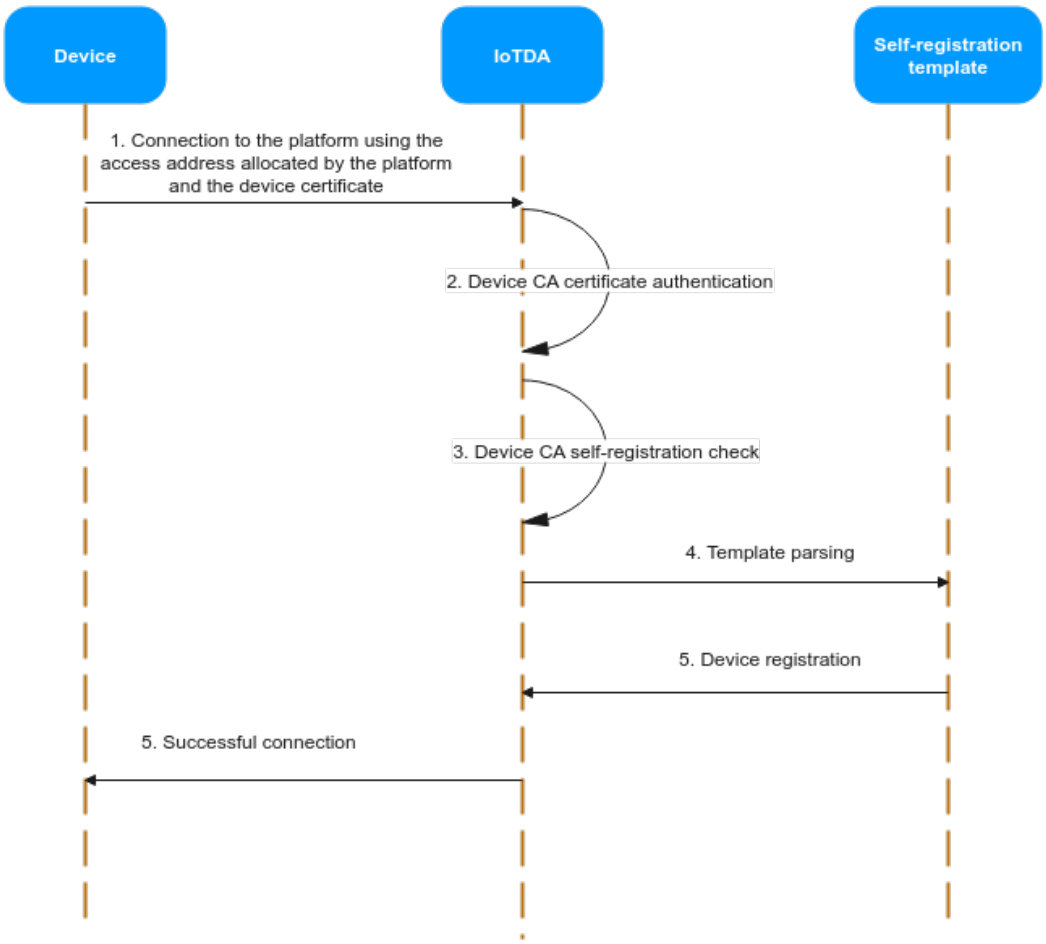
- **Obtain the Device CA Certificate List**
- **Upload a Device CA Certificate**
- **Delete a Device CA Certificate**
- **Verify a Device CA Certificate**

3.3.4 Device Self-Registration

Introduction

The device self-registration function enables automatic registration of a device with the IoT platform upon initial connection, eliminating the need for pre-registration on the console. This process is facilitated through certificate authentication, with device information stored in device certificates.

Figure 3-119 Device self-registration process



Scenarios

- Device requirements: Some devices cannot be pre-registered on the console or by calling APIs.
- Internet of Vehicles (IoV): When a head unit is started, it automatically registers with the platform and reports data to the platform, simplifying the development of the vehicle application.
- Multi-instance scenario: Enterprise customers can utilize the self-registration function to efficiently manage devices across multiple IoTDA instances. This eliminates the need to provision and register devices separately on each instance beforehand.

Constraints

- Max. self-registration templates for an instance: 10.
- To use the device self-registration function, the device must use transport layer security (TLS) and enable the **Server Name Indication (SNI)** extension. The SNI must carry the domain name allocated by the platform. To obtain the domain name, see **How Do I Obtain the Platform Access Address?**
- Only available for MQTTS certificate authentication.
- Not available for standard edition instances in the CN East-Shanghai1 region.

NOTICE

Devices registered through self-registration are authenticated based on the self-registration template. Modifying or disabling the self-registration template may affect device authentication. Exercise caution.

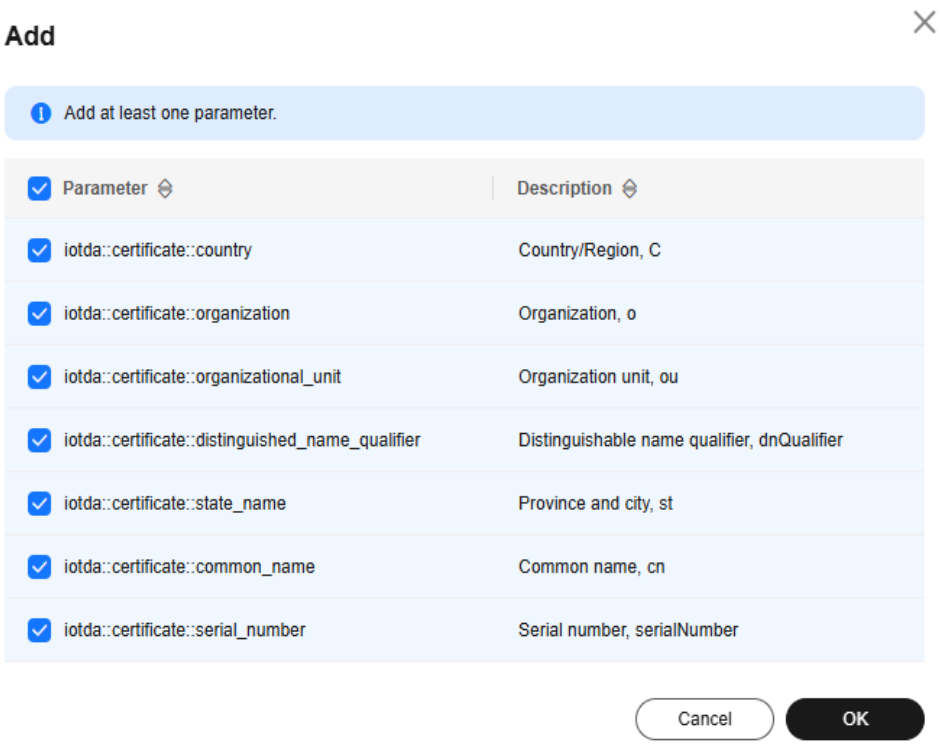
Procedure

Step 1 Access the IoTDA service page and click **Access Console**. Click the target instance card.

Step 2 Create a product.

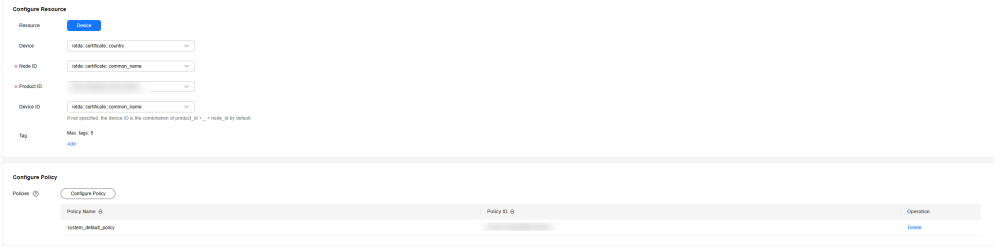
Step 3 In the navigation pane, choose **Devices** > **Self-registration Template**. Click **Create Template**. On the displayed page, enter basic information, and click the button for adding the required parameters to the template.

Figure 3-120 Self-registration template - Adding parameters



Step 4 Select the device name, node ID, device ID, and product ID in the resource configuration area.

Figure 3-121 Self-registration template - Creating a template



NOTE

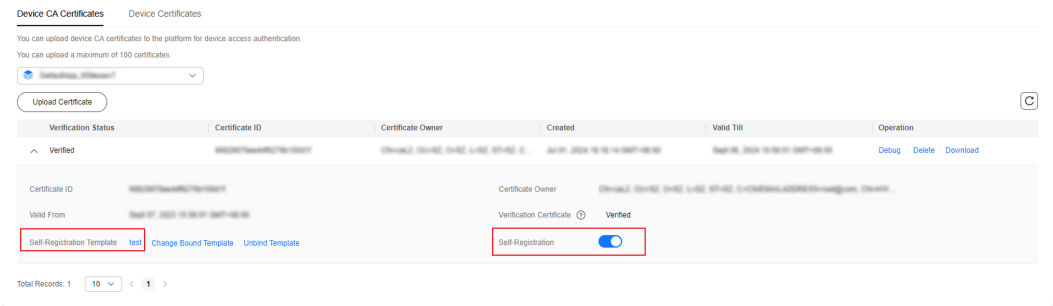
The platform predefines the parameters that can be declared and referenced in the template, as shown below. The certificate must contain the parameters referenced in the template.

- **iotda::certificate::country**: country
- **iotda::certificate::organization**: organization
- **iotda::certificate::organizational_unit**: department
- **iotda::certificate::distinguished_name_qualifier**: distinguished name
- **iotda::certificate::state_name**: province/state
- **iotda::certificate::common_name**: common name
- **iotda::certificate::serial_number**: serial number

Step 5 Add a policy in the policy configuration area. The added policy is automatically bound to the device during self-registration. For details, see **Device Topic Policies**.

Step 6 In the navigation pane, choose **Devices** > **Device Certificates**. Create a device certificate by referring to **Registering a Device Authenticated by an X.509 Certificate**. Upload the CA certificate to the platform for verification, bind the self-registration template created in **Step 3**, and enable the self-registration function.

Figure 3-122 Device CA certificate - Binding a template

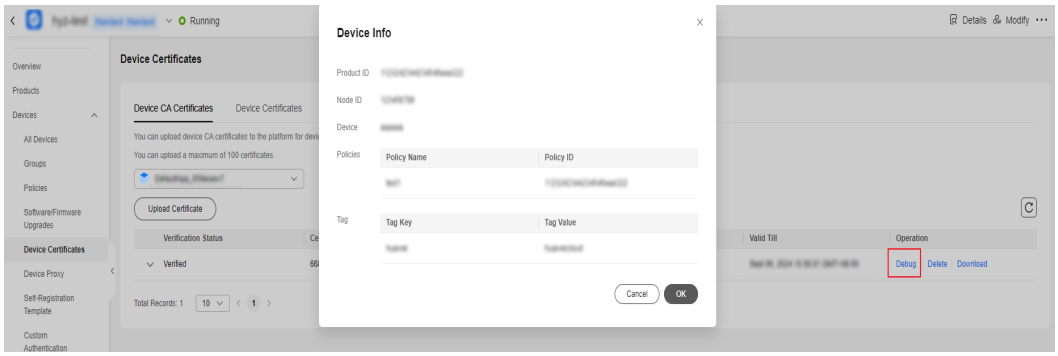


NOTE

The CA certificate and the product associated with the product ID in the template must be in the same resource space.

Step 7 On the device CA certificate tab page, click **Debug**, upload the device certificate created in **Step 6**, and check whether the pre-parsed device information meets your expectation.

Figure 3-123 Device CA certificate - Debugging a certificate



----End

Simulator-based Verification

Use MQTT.fx to simulate a device to access the platform for automatic device registration.

Step 1 Download **MQTT.fx** (64-bit OS) or **MQTT.fx** (32-bit OS) and install it.

Step 2 Open MQTT.fx, set connection parameters by referring to **Table 3-23**, and click **Apply**.

Table 3-23 Connection parameters

Parameter	Description
Broker Address	Platform access address (see **How Do I Obtain the Platform Access Address?**)
Broker Port	8883
Client ID	Any string. Recommended: Set this parameter according to the platform rules in **Device Connection Authentication** to ensure continued access to the platform via certificate authentication even after the template is disabled.
User Name	Any string. Recommended: Set this parameter according to the platform rules in **Device Connection Authentication** to ensure continued access to the platform via certificate authentication even after the template is disabled.
Password	Empty
Enable SSL/TLS	True
Self signed certificates	True
CA File	Platform CA certificate (see **Certificates**)
Client Certificate File	Path of the device certificate file

Parameter	Description
Client Key File	Path of the private key file of the device certificate
Client Key Password	Private key password (not necessary if there is no password)

Figure 3-124 MQTT.fx Settings

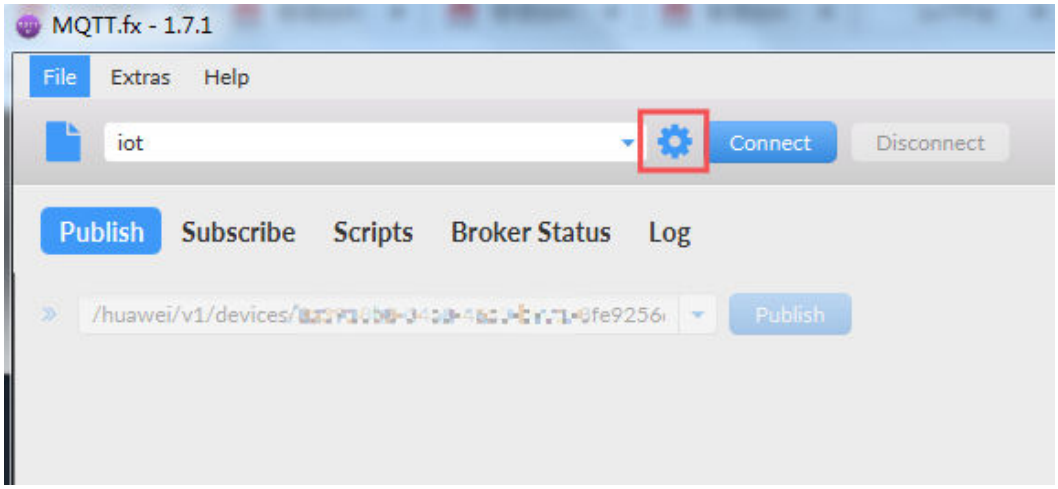


Figure 3-125 Connection parameters

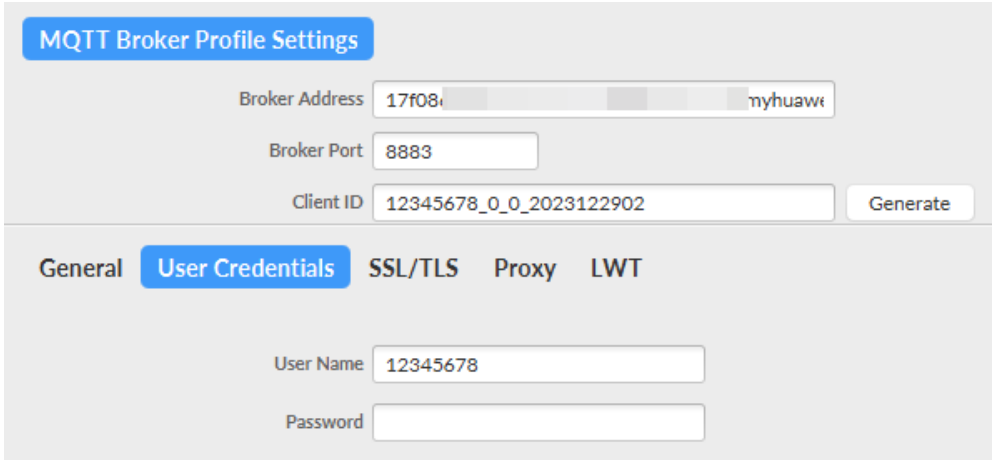
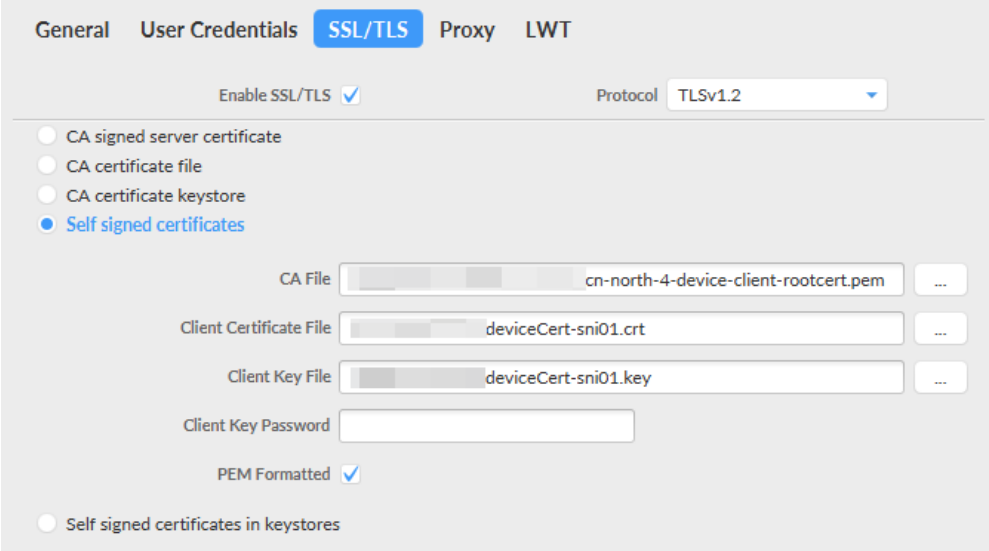
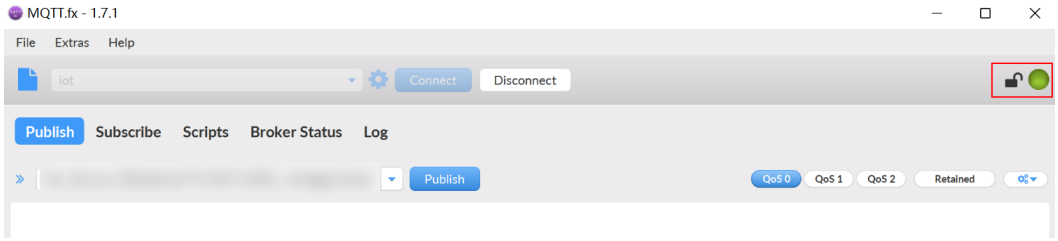


Figure 3-126 Certificate information



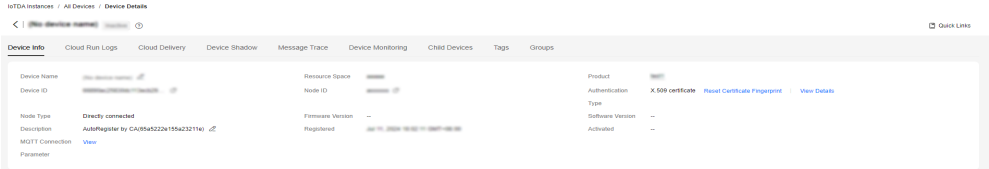
Step 3 Click **Connect**. If the icon in the upper right corner turns green, the simulated device has been authenticated and connected.

Figure 3-127 Device simulator connected



Step 4 In the navigation pane of the IoTDA console, choose **Devices** > **All Devices**. On the device list tab page, search for the device by device ID or node ID. The device is displayed as registered and online.

Figure 3-128 Device - Self-registered device details



----End

3.4 Device SDK Access

Huawei Cloud IoTDA, a platform for access and management of a large number of devices, allows you to connect your physical devices to the cloud, where you can collect device data and deliver commands to devices for remote control. It can also work with other Huawei Cloud services to help you quickly develop IoT solutions.

This section describes how to efficiently connect devices to IoTDA using the Java SDK, with a custom gas meter product model as an example. It covers developing SDK code for reporting device data (messages and properties) and delivering commands for remote configuration and control, as well as integrating the code into devices.

NOTE

Device SDKs are not exclusive to a specific product model. You can customize the code based on the site requirements.

Device SDKs

Table 3-24 Device SDKs

Resource Package	Description	Download Link
IoT Device Java SDK	The demo provides the code sample for calling the SDK APIs. For details, see **IoT Device Java SDK**.	**IoT Device Java SDK**
IoT Device C SDK for Linux/Windows	The demo provides the code sample for calling the SDK APIs. For details, see **IoT Device C SDK for Linux/Windows**.	**IoT Device C SDK for Linux/Windows**
IoT Device C# SDK	The demo provides the code sample for calling the SDK APIs. For details, see **IoT Device C# SDK**.	**IoT Device C# SDK**
IoT Device Android SDK	The demo provides the code sample for calling the SDK APIs.	**IoT Device Android SDK**
IoT Device Go SDK (Community Edition)	The demo provides the code sample for calling the SDK APIs.	**IoT Device Go SDK (Community Edition)**
IoT Device Python SDK	The demo provides the code sample for calling the SDK APIs.	**IoT Device Python SDK**
IoT Device Tiny C SDK for Linux/Windows	The demo provides the code sample for calling the SDK APIs.	**IoT Device Tiny C SDK for Linux/Windows**

Resource Package	Description	Download Link
IoT Device ArkTS (OpenHarmony) SDK	The demo provides the code sample for calling the SDK APIs.	IoT Device ArkTS (OpenHarmony) SDK

Table 3-25 SDK functions

Function	C	Java	C#	Android	Go	Python	C Tiny	ArkTS
Property reporting	√	√	√	√	√	√	√	√
Message reporting and delivery	√	√	√	√	√	√	√	√
Event reporting and delivery	√	√	√	√	√	√	√	×
Command delivery and response	√	√	√	√	√	√	√	√
Device shadow	√	√	√	√	√	√	√	√
OTA upgrade	√	√	√	√	√	√	√	×
Bootstrap	√	√	√	√	√	√	√	×
Time synchronization	√	√	√	√	√	√	√	×
Gateway and child device management	√	√	√	√	√	√	√	×
Device-side rule engine	√	×	√	×	×	×	√	×
Remote secure shell (SSH)	√	×	√	×	×	×	×	×
Anomaly detection	√	×	√	×	×	×	×	×

Function	C	Java	C#	Android	Go	Python	C Tiny	ArkTS
Device-cloud secure communication (soft bus)	√	×	√	×	×	×	×	×
Machine-to-machine (M2M) function	√	×	√	×	×	×	×	×
Generic-protocol access	√	√	√	√	×	√	×	×

Prerequisites

- Development environment: The integrated environment (IntelliJ IDEA) of Java has been installed, and the environment such as Maven has been configured.
- This example uses the MQTTS protocol on the device.
- You have registered a Huawei Cloud account and completed real-name authentication.
- You have subscribed to **IoTDA** on the console.

Service Flow

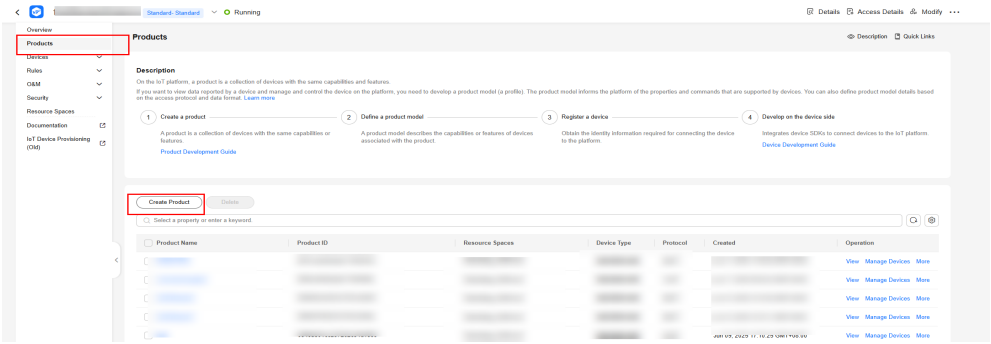
Use the IoT Device Java SDK to connect devices to IoTDA and report data and deliver commands.

1. **Product creation**: Create an MQTT product.
2. **Product model development**: Create a product model for a gas meter on the platform that allows for remote reporting of readings and the delivery of configurations and commands.
3. **Device registration**: Register a device using the MQTT protocol.
4. **Access to Huawei Cloud via Java SDK**: Download the SDK, adapt the code, and use the Java SDK to activate the device registered on the platform.
5. **Message reporting**: Adapt the code and use the Java SDK to report messages to the platform.
6. **Property reporting**: Adapt the code and use the Java SDK to report device properties to the platform.
7. **Command delivery**: Adapt the code and deliver commands on the console to set device properties remotely.
8. **Java SDK integration and running**: Package the adapted SDK into a runnable file, import the file to the IoT device, and run the file to connect the device to IoTDA.

Creating a Product

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card. Choose **Products** in the navigation pane and click **Create Product**.

Figure 3-129 Creating a product

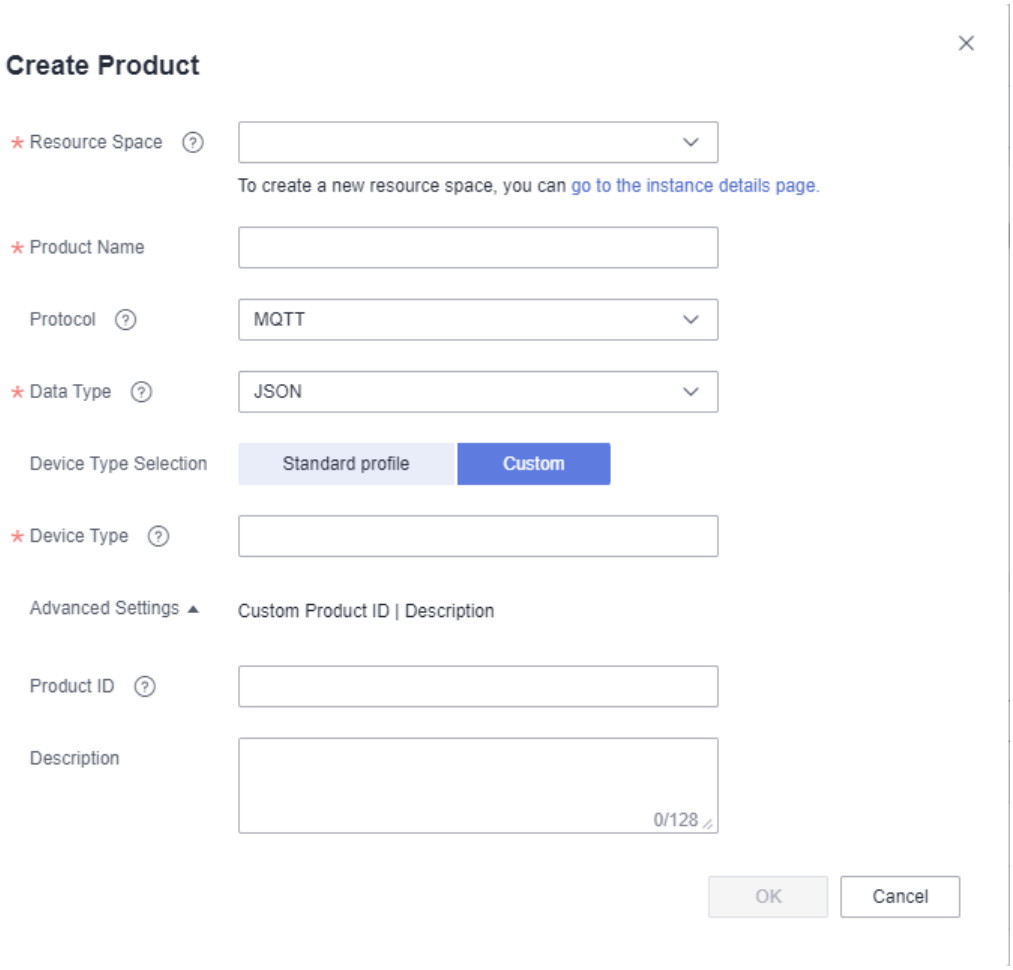


Step 2 Create a product whose protocol type is MQTT and device type is custom gas meter. Set parameters by referring to the following table and click **OK**.

Table 3-26 Parameters for creating a product

Resource Space	Select the default resource space.
Product Name	Customize a product name, for example, **Gas Meter**.
Protocol	Select **MQTT**.
Data Type	Select **JSON**.
Device Type Selection	Select **Custom**.
Device Type	Customize a device type, for example, **Custom Gas Meter**.

Figure 3-130 Creating an MQTT product



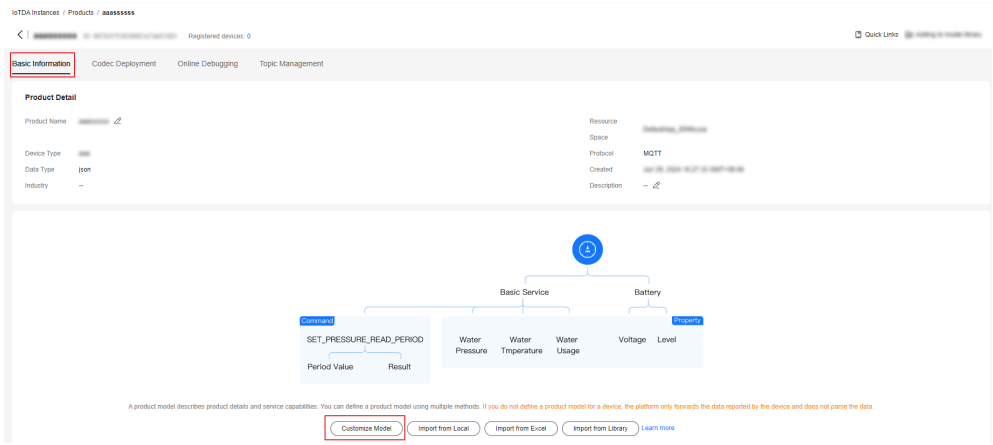
----End

Developing a Product Model

Step 1 Click the created product to access its details page.

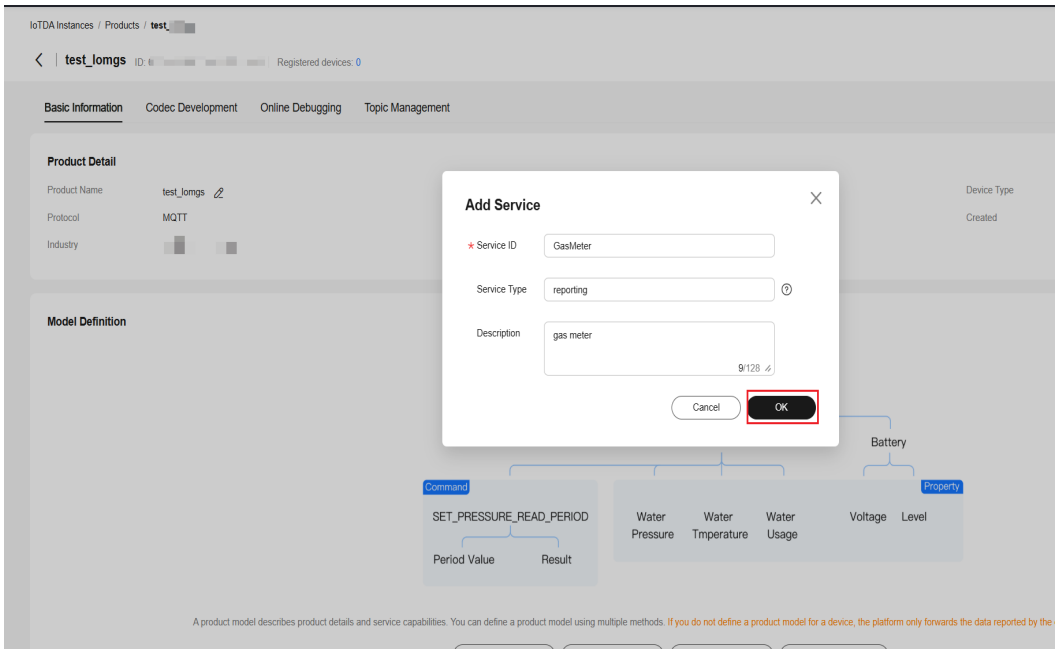
Step 2 On the **Basic Information** tab page, click **Customize Model** to add services of the product.

Figure 3-131 Custom model - MQTT



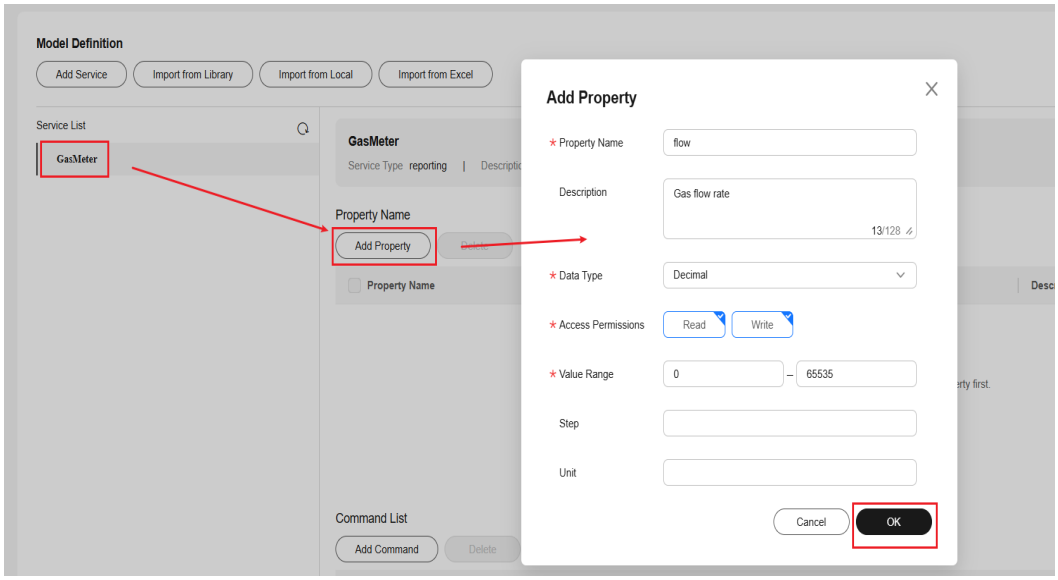
Step 3 On the displayed **Add Service** page, enter the service ID, service type, and service description, and click **OK**.

Figure 3-132 Adding a service - GasMeter



Step 4 Choose **GasMeter** in the service list, click **Add Property**, set parameters according to the following figure, and click **OK**.

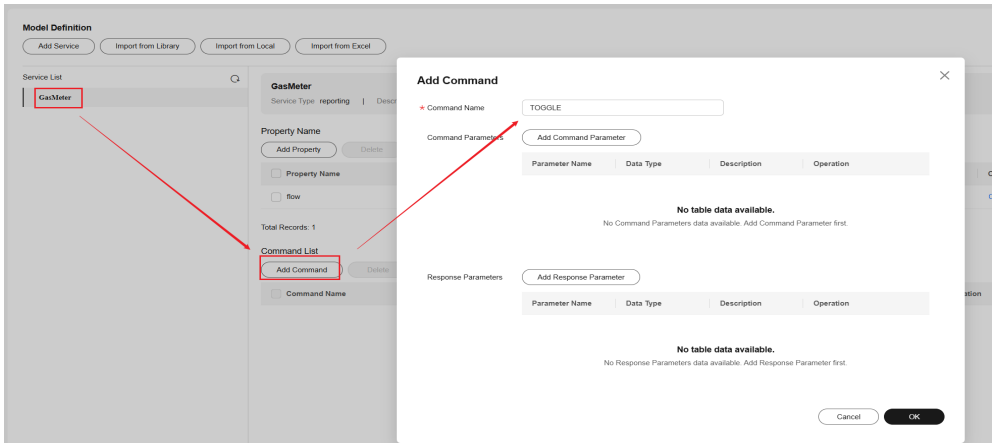
Figure 3-133 Adding a property - flow



Step 5 Add a command model.

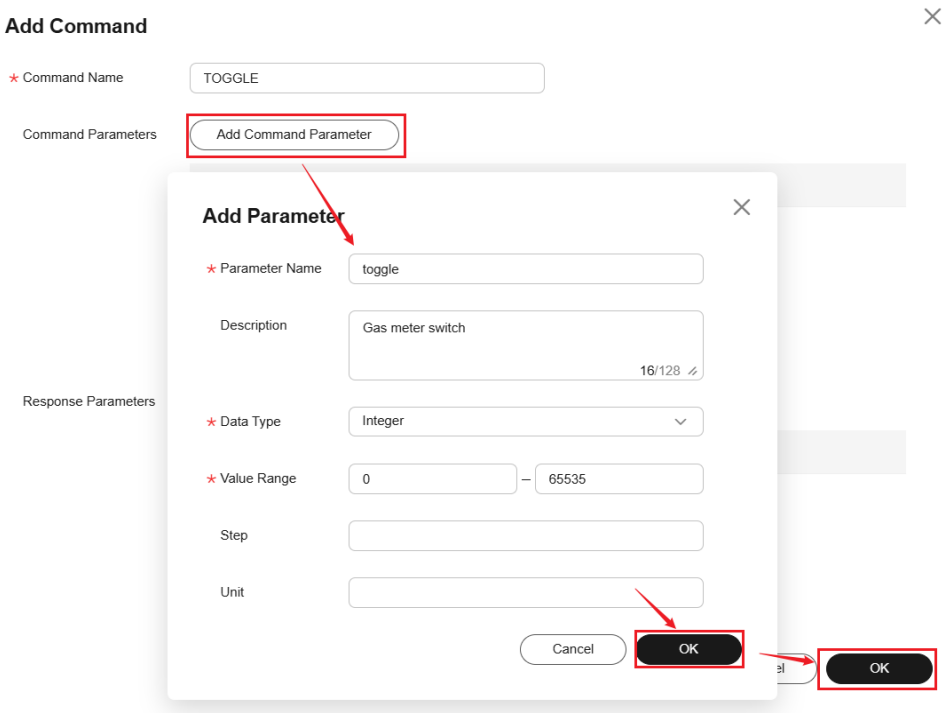
1. Choose **GasMeter** in the service list, click **Add Command**, and set the command name to **TOGGLE**.

   Figure 3-134 Adding a command - TOGGLE

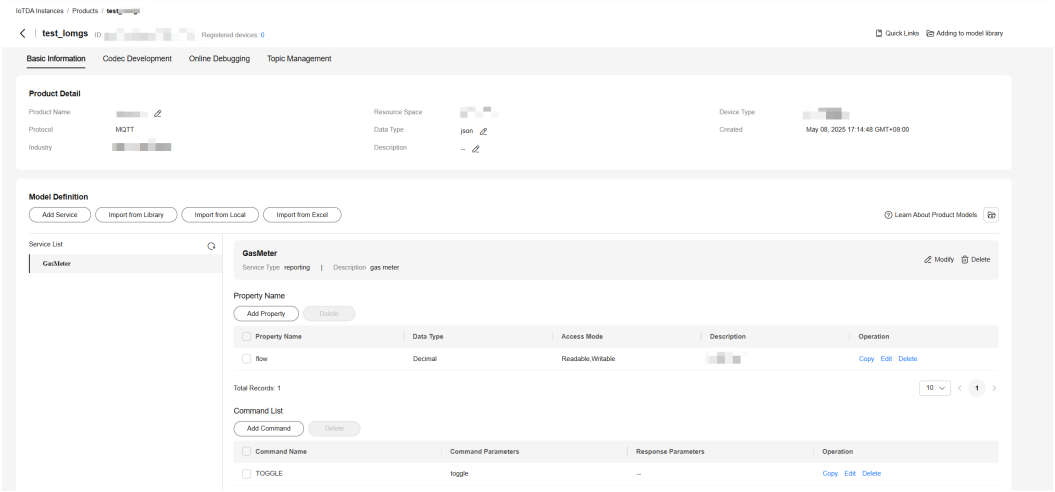
2. On the displayed page, click **Add Command Parameter**, set parameters according to the following figure, and click **OK**.

Figure 3-135 Adding a command - toggle



Step 6 Check the product model details.

Figure 3-136 Product model - gas meters

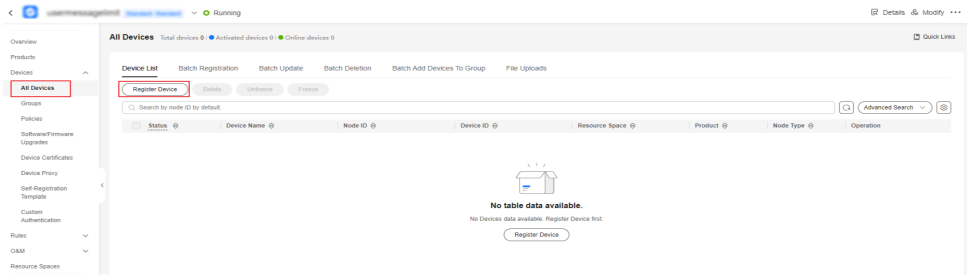


----End

Registering a Device

Step 1 On the IoTDA console, click the target instance card. In the navigation pane, choose **Devices** > **All Devices**. Click **Register Device**.

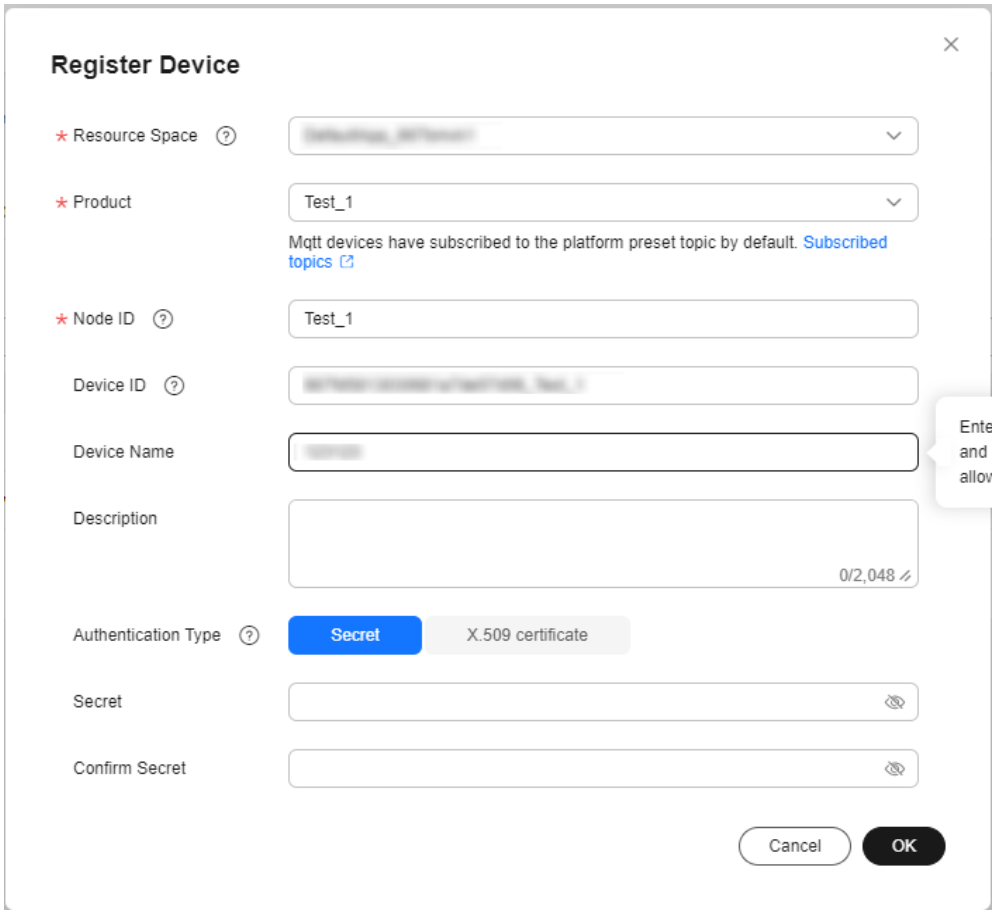
Figure 3-137 Registering a device



Step 2 Set the parameters as prompted and click **OK**.

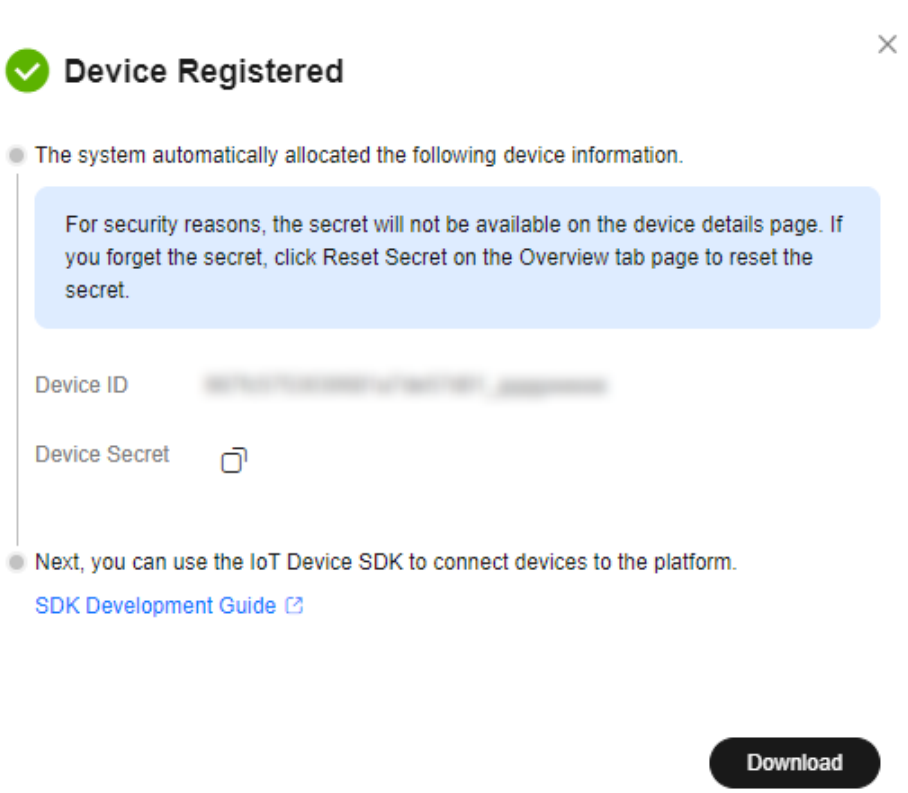
Parameter	Description
Resource Space	Ensure that the device and its associated product belong to the same resource space.
Product	Select a corresponding product.
Node ID	Customize a unique physical identifier for the device. Enter 4 to 64 characters. Use only letters, digits, underscores (_), and hyphens (-).
Device Name	Customize the device name.
Authentication Type	Select **Secret**.
Secret	If you do not set this parameter, IoTDA automatically generates a value.

Figure 3-138 Registering a device - MQTT



Step 3 After the device is registered, the platform automatically generates a device ID and secret. Save the device ID and secret for device access.

Figure 3-139 Device - Device registered



----End

Accessing to Huawei Cloud Using Java SDK

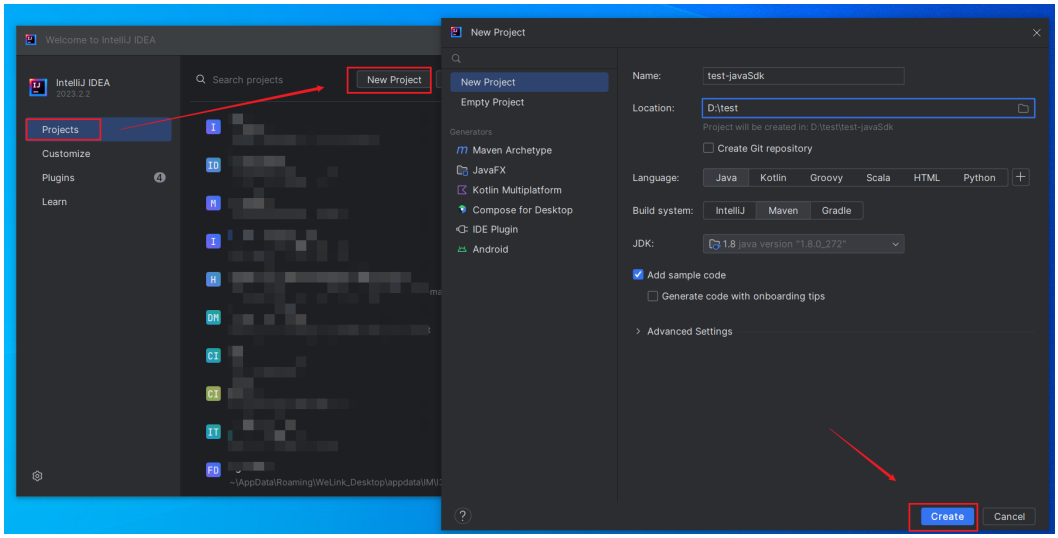
You can use IntelliJ IDEA (IntelliJ IDEA 2023 Community Edition is used as an example) to write and debug code. Ensure that the IDEA environment is normal and Maven is available. For details about the Java SDK usage and APIs, see **README**.

NOTE

You are advised to adapt the SDK code and activate the device on the computer, and then import the modified code to the device for integration.

Step 1 Create a project. Open IntelliJ IDEA and click **New Project**. On the displayed page, enter the project name and project path, set **Language** to **Java**, set **Build System** to **Maven**, and click **Create**.

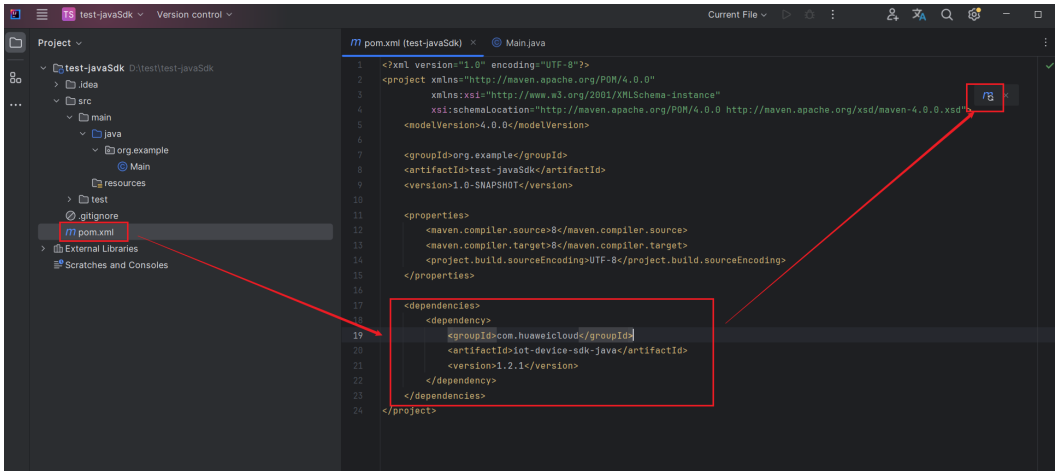
Figure 3-140 Creating a Java project



Step 2 Add the Maven reference. After the project is created, the **pom.xml** and **Main.java** files are automatically generated in the project. Open the **pom.xml** file, add the Maven reference of the Java SDK, and click the Maven update icon in the upper right corner. If an error is reported, check the Maven configuration.

```
<dependencies>
    <dependency>
        <groupId>com.huaweicloud</groupId>
        <artifactId>iot-device-sdk-java</artifactId>
        <version>1.2.1</version>
    </dependency>
</dependencies>
```

Figure 3-141 Adding the Maven reference of the Java SDK



Step 3 Write the reference code to establish a connection with the device.

1. Open the **Main.java** file and copy the following code to the file:

```
import com.huaweicloud.sdk.iot.device.IoTDevice;
import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.nio.file.Files;
import static java.nio.file.StandardCopyOption.REPLACE_EXISTING;
public class Main {
    private static final String IOT_ROOT_CA_RES_PATH = "ca.jks";
```



```
    private static final String IOT_ROOT_CA_TMP_PATH = "huaweicloud-iotda-tmp-" +
IOT_ROOT_CA_RES_PATH;
    public static void main(String[] args) throws InterruptedException, IOException {
        // Load the CA certificate of the IoT platform for server verification.
        File tmpCAFile = new File(IOT_ROOT_CA_TMP_PATH);
        try (InputStream resource =
Main.class.getClassLoader().getResourceAsStream(IOT_ROOT_CA_RES_PATH)) {
            Files.copy(resource, tmpCAFile.toPath(), REPLACE_EXISTING);
        }
        // Create a device and initialize it. Replace the access address with your own address.
        IoTDevice device = new IoTDevice("ssl://xxx.st1.iotda-device.cn-
north-4.myhuaweicloud.com:8883",
                "5e06bfee334dd4f33759f5b3_demo", "mysecret", tmpCAFile);
        if (device.init() != 0) {
            return;
        }
    }
}
```

2. Add a certificate file. **Obtain the CA certificate on the device side** based on your region and change the certificate name to **ca.jks**. Save the certificate to the **resources** directory of the project.

   Figure 3-142 Obtaining the CA file on the device side

   Table 1 Certificates

Certificate Package Name	Region and Edition	Certificate Type	Certificate Format	Description	Download Link
certificate	CN-Hong Kong, AP-Singapore, AP-Bangkok, AP-Jakarta, AF-Johannesburg, LA-Santiago, LA-Sao Paulo1, LA-Mexico City2, and ME-Riyadh	Device certificate	pem, jks, and bks	Used by a device to verify the platform identity. The certificate must be used together with the device access domain name.	Certificate file
certificate	CN-Hong Kong, AP-Singapore, AP-Bangkok, AP-Jakarta, AF-Johannesburg, LA-Santiago, LA-Sao Paulo1, LA-Mexico City2, and ME-Riyadh	Application certificate	pem, jks, and bks	Application access: HTTPS/AMQPS/MQTTS platform CA certificates	Certificate file

3. Modify access information. Change the device access address, device ID, and device secret to those obtained in **Registering a Device**.

Figure 3-143 Modifying device connection parameters

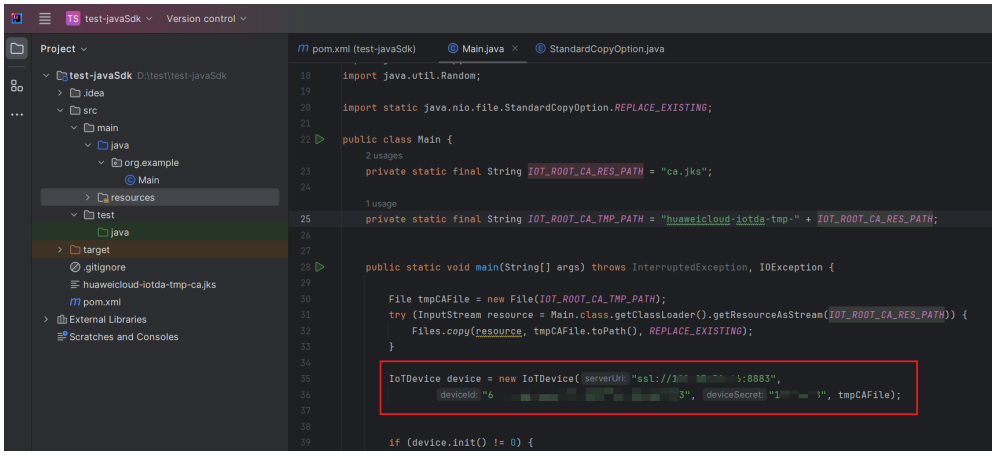
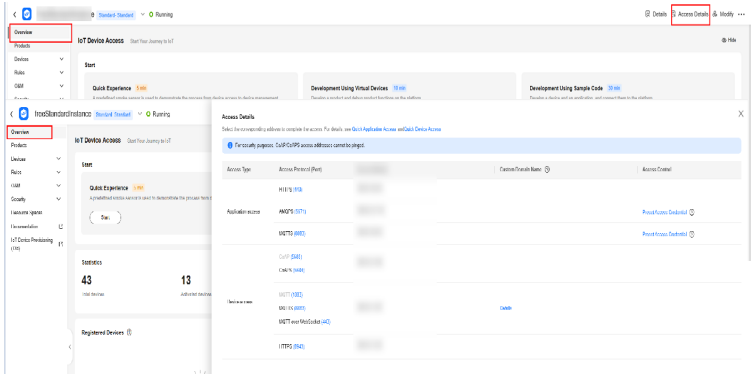


Figure 3-144 Obtaining access information



NOTE

- Select the MQTTS access address and copy it to the code. To use the MQTT protocol, change **ssl://xxx.st1.iotda-device.cn-north-4.myhuaweicloud.com:8883** to **tcp://xxx.st1.iotda-device.cn-north-4.myhuaweicloud.com:1883**.
- When you create a device, the system displays a dialog box asking you whether to save the device ID and key. If you choose to save, the device ID and key are saved as a file on your computer.

Step 4 Write and run the code. Click the run button in IDEA. The device is online on the platform.

Figure 3-145 Running code in IDEA

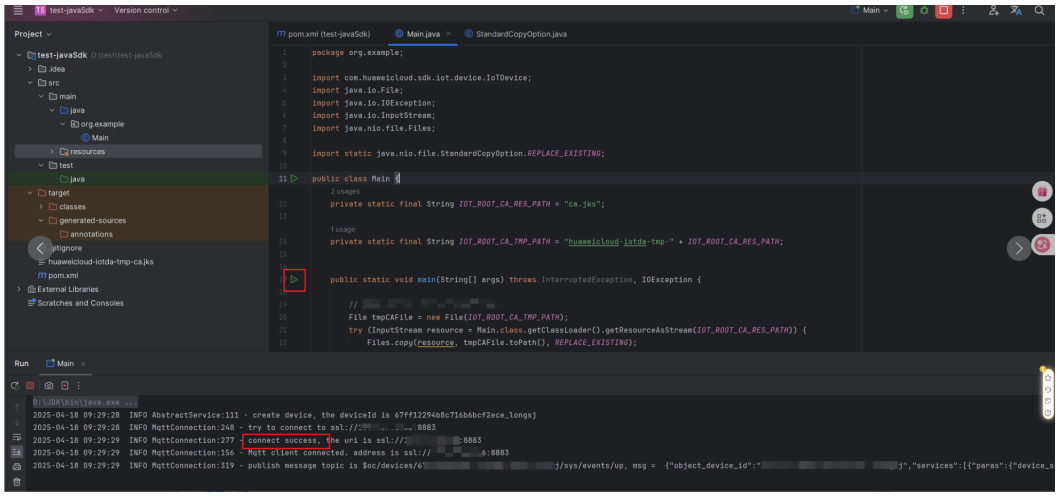
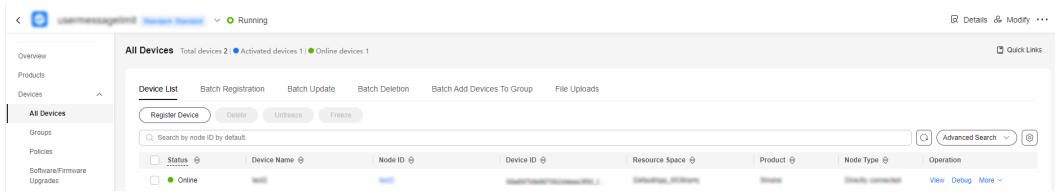


Figure 3-146 Device online status



----End

NOTE

The SDK automatically reconnects to the device if the device is disconnected due to network problems.

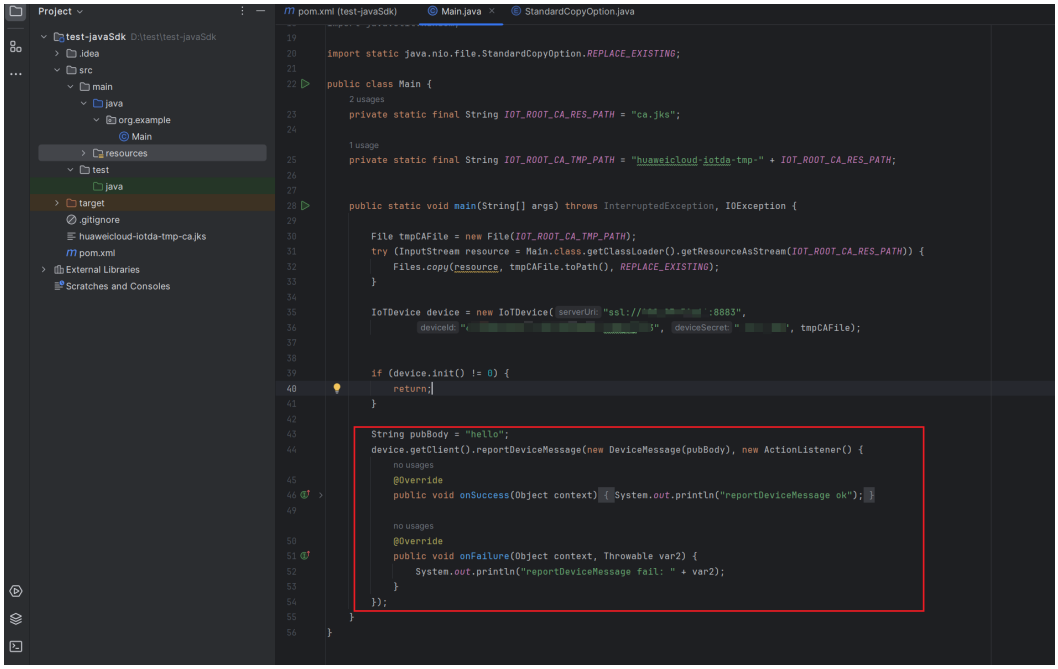
Reporting a Message

Develop the message reporting function by referring to **the guide for message reporting and delivery**.

Step 1 Copy the following code to the **Main.java** file of the new project. Put the code after the **device.init()** method, which indicates that the connection is successfully established.

```
// pubBody indicates the message to be reported. It will be edited into the standard format for reporting data.
// The default topic reported by the reportDeviceMessage method is $oc/devices/{device_id}/sys/messages/up.
String pubBody = "hello";
device.getClient().reportDeviceMessage(new DeviceMessage(pubBody), new ActionListener() {
    @Override
    public void onSuccess(Object context) {
        System.out.println("reportDeviceMessage ok");
    }
    @Override
    public void onFailure(Object context, Throwable var2) {
        System.out.println("reportDeviceMessage fail: " + var2);
    }
});
```

Figure 3-147 Reporting messages - IDEA



Step 2 Enable device message tracing to check message records. Locate the device on the Huawei Cloud console, go to device details page, and choose **Message Trace** > **Start Trace**. Run the code again to check the reported messages.

Figure 3-148 Device list - Viewing details

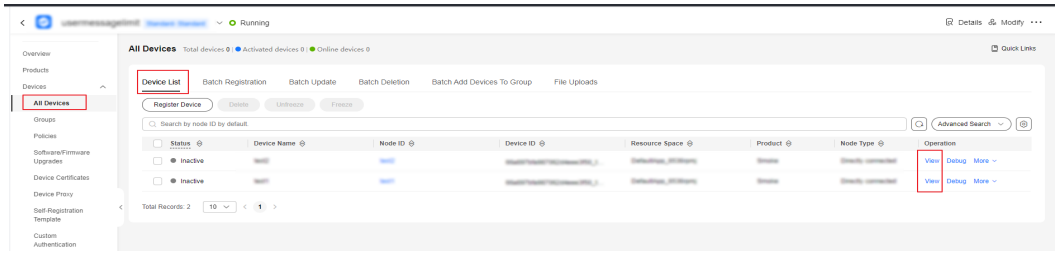
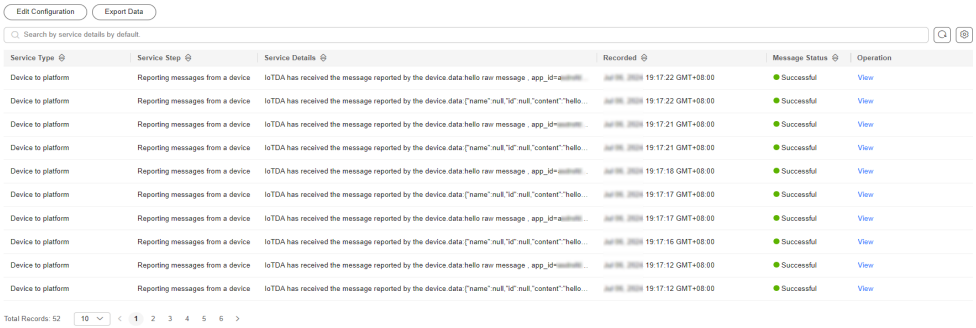


Figure 3-149 Message tracing - Viewing device_sdk_java tracing result



----End

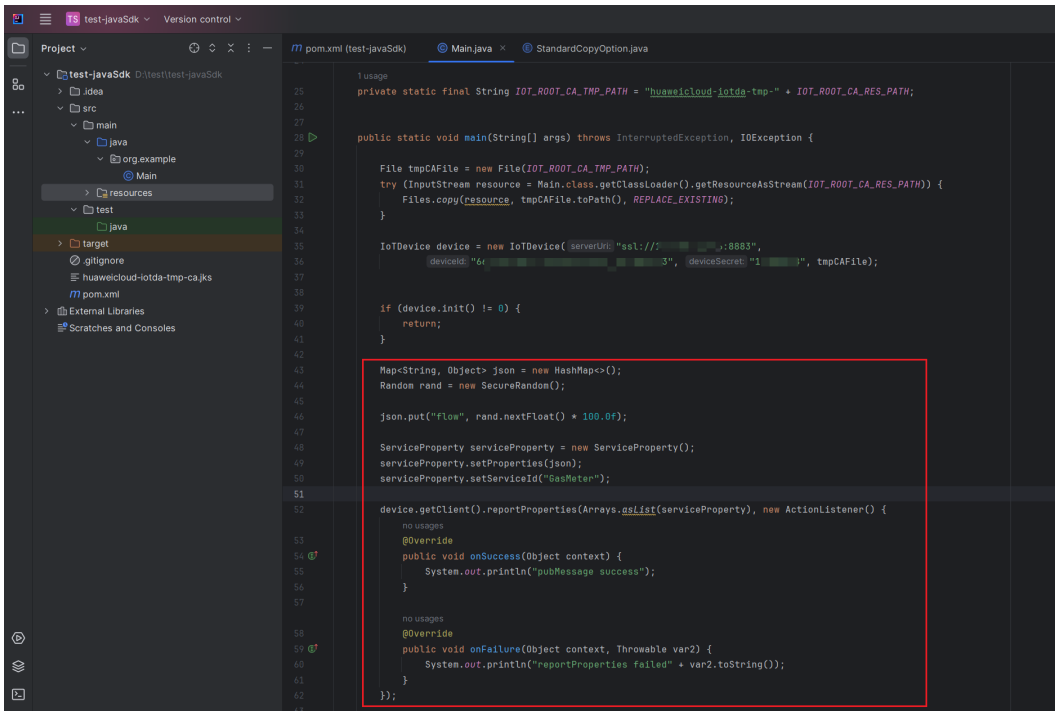
Reporting Properties

Develop the property reporting function by referring to **the guide for property reporting**.

Step 1 Copy the following code to the **Main.java** file. Put the code after the **device.init()** method, which indicates that the connection is successfully established.

```
// Report properties.
Map<String, Object> json = new HashMap<>();
Random rand = new SecureRandom();
// Set properties based on the product model.
json.put("flow", rand.nextFloat() * 100.0f);
ServiceProperty serviceProperty = new ServiceProperty();
serviceProperty.setProperties(json);
serviceProperty.setServiceId("GasMeter"); // serviceId must be the same as that in the product model.

device.getClient().reportProperties(Arrays.asList(serviceProperty), new ActionListener() {
    @Override
    public void onSuccess(Object context) {
        System.out.println("pubMessage success");
    }
    @Override
    public void onFailure(Object context, Throwable var2) {
        System.out.println("reportProperties failed" + var2.toString());
    }
});
```

Figure 3-150 Reporting properties - IDEA



NOTE

The gas meter is used as an example. You can adapt the code as required.

Step 2 Check the reported property value on the platform. On the Huawei Cloud console, locate the target device to access its details page, click the **Device Info** tab, and run the code again. The reported property value is displayed.

Figure 3-151 Device list - Viewing details

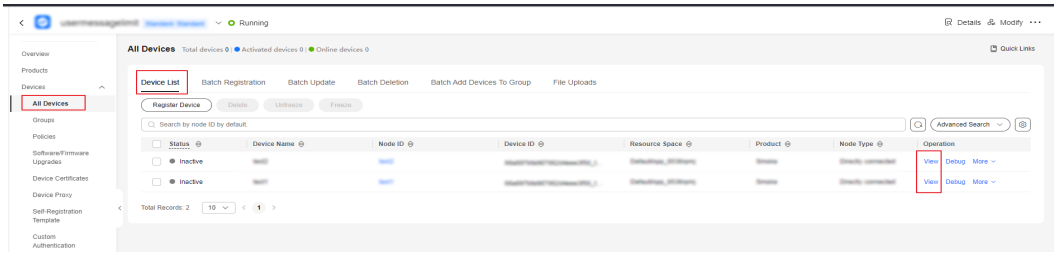
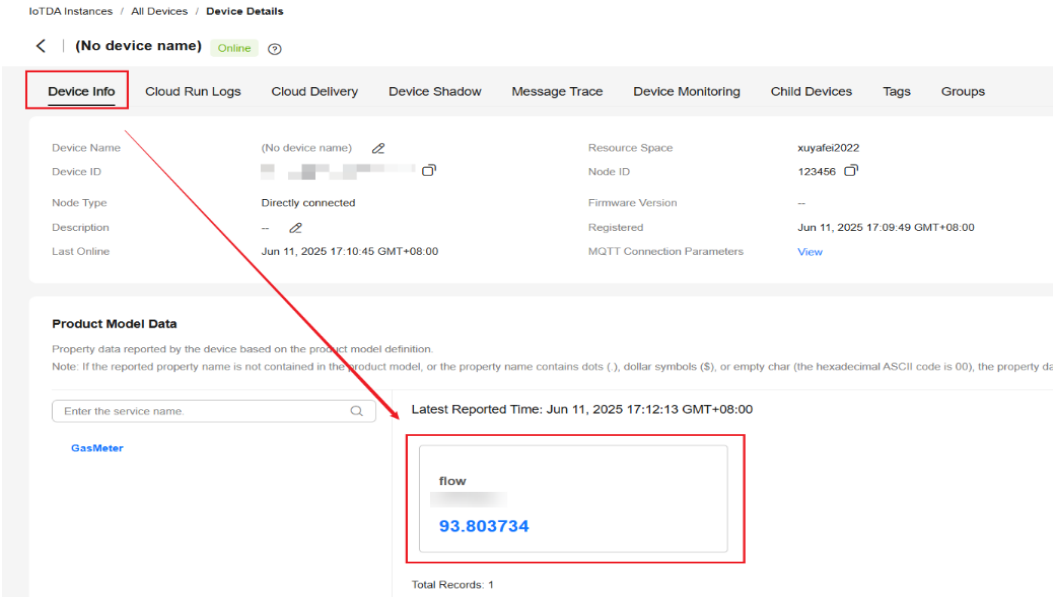


Figure 3-152 Viewing reported data - Flow



----End

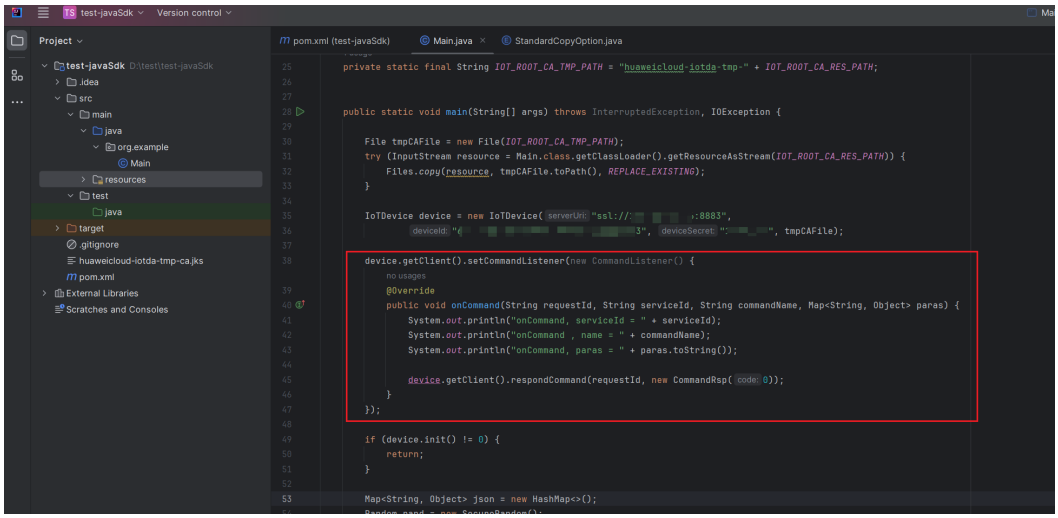
Delivering Commands

Develop the delivery function by referring to **the guide for command delivery**.

Step 1 Copy the following code to the **Main.java** file. Put the code before the **device.init()** method, which is used to establish a link connection.

```
// Set the listener to receive downstream data.
device.getClient().setCommandListener(new CommandListener() {
    @Override
    public void onCommand(String requestId, String serviceId, String commandName, Map<String, Object>
paras) {
        System.out.println("onCommand, serviceId = " + serviceId);
        System.out.println("onCommand , name = " + commandName);
        System.out.println("onCommand, paras = " + paras.toString());
        // Process the command.

        // Send the command response.
        device.getClient().respondCommand(requestId, new CommandRsp(0));
    }
});
```


Figure 3-153 Command delivery - IDEA



Step 2 Deliver a command on the platform. Run the SDK code. On the Huawei Cloud console, locate the target device to access its details page, click the **Cloud Delivery** tab and then the **Command Delivery** tab, click **Deliver Command**, and click **OK**. The delivered value is received on the IDEA console.

Figure 3-154 Device list - Viewing details

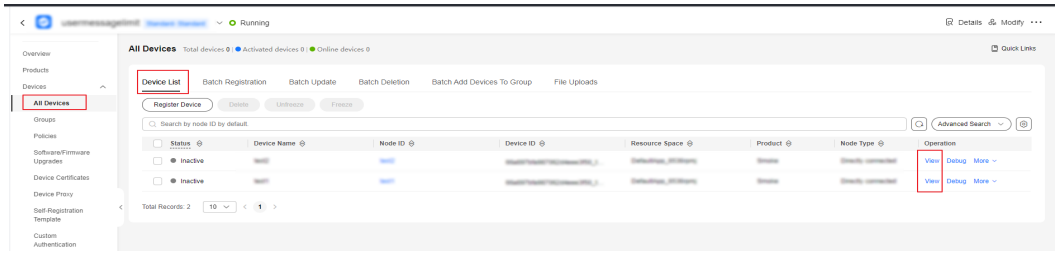
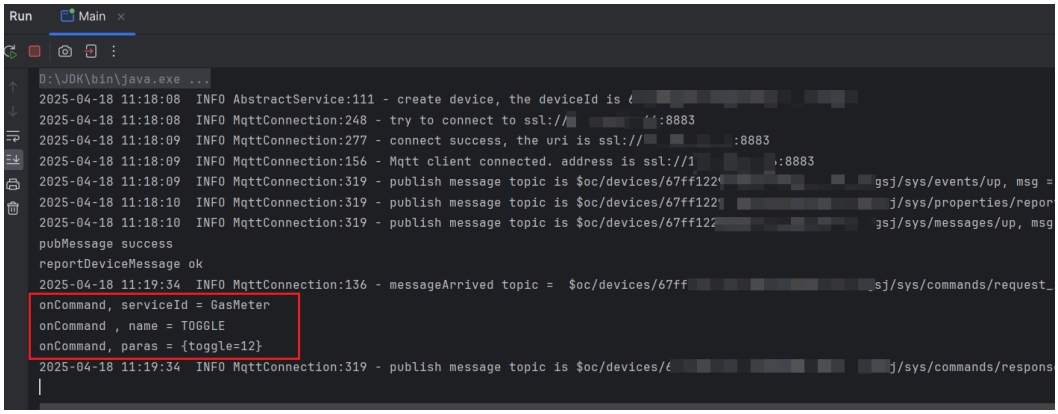


Figure 3-155 Checking the IDEA command delivery structure



----End

Integrating and Running the Device Java SDK

Huawei Cloud IoT device SDKs seamlessly integrate with IoT devices, as demonstrated in this section using a Linux-based IoT device and the Java SDK. By

debugging and transferring the Java SDK-generated JAR package from IDEA to the Linux device, you can efficiently connect the device to IoTDA.

Prerequisites

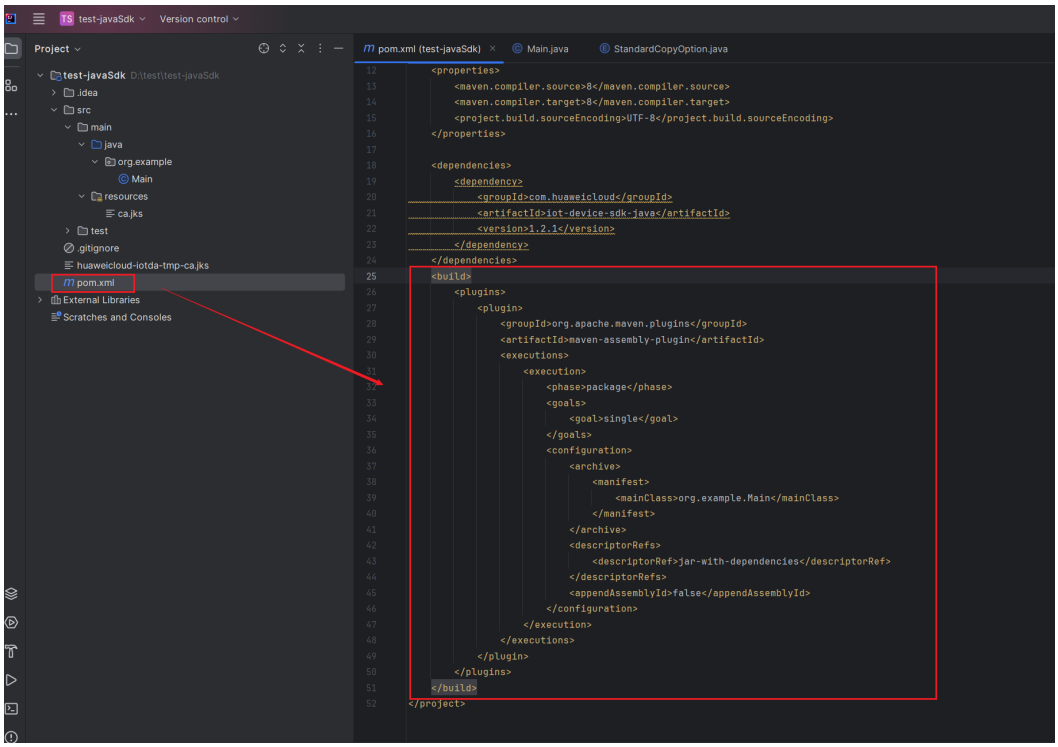
The device runs Linux and JDK has been installed.

Procedure

Step 1 Copy the configuration below to the **pom.xml** file in the root directory.

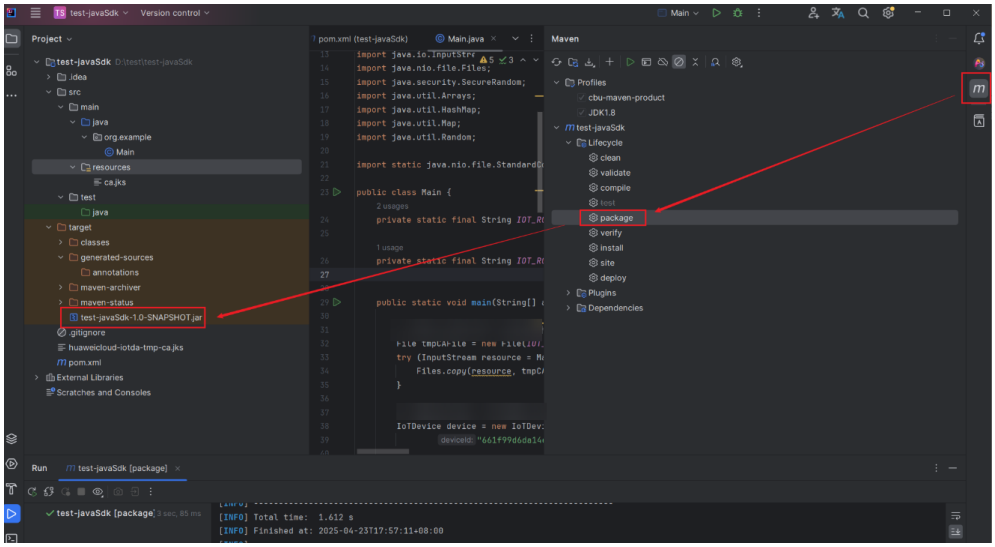
```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                    <configuration>
                        <archive>
                            <manifest>
                                <mainClass>org.example.Main</mainClass>
                            </manifest>
                        </archive>
                        <descriptorRefs>
                            <descriptorRef>jar-with-dependencies</descriptorRef>
                        </descriptorRefs>
                        <appendAssemblyId>false</appendAssemblyId>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Figure 3-156 IDEA - Adding the package parameter



Step 2 Open Maven in IDEA and click **package** to generate a JAR package in the directory.

Figure 3-157 IDEA - Generating a JAR package



Step 3 Copy the JAR package to the Linux device and run the **java -jar test-javaSdk-1.0-SNAPSHOT** command. If the code is successfully executed, the device is successfully connected to IoTDA.

Figure 3-158 Running the JAR package in Linux

NOTE

If no Java command is displayed, run the **java -version** command to check whether JDK is installed and the JDK version.

----End

3.5 MQTT(S) Access

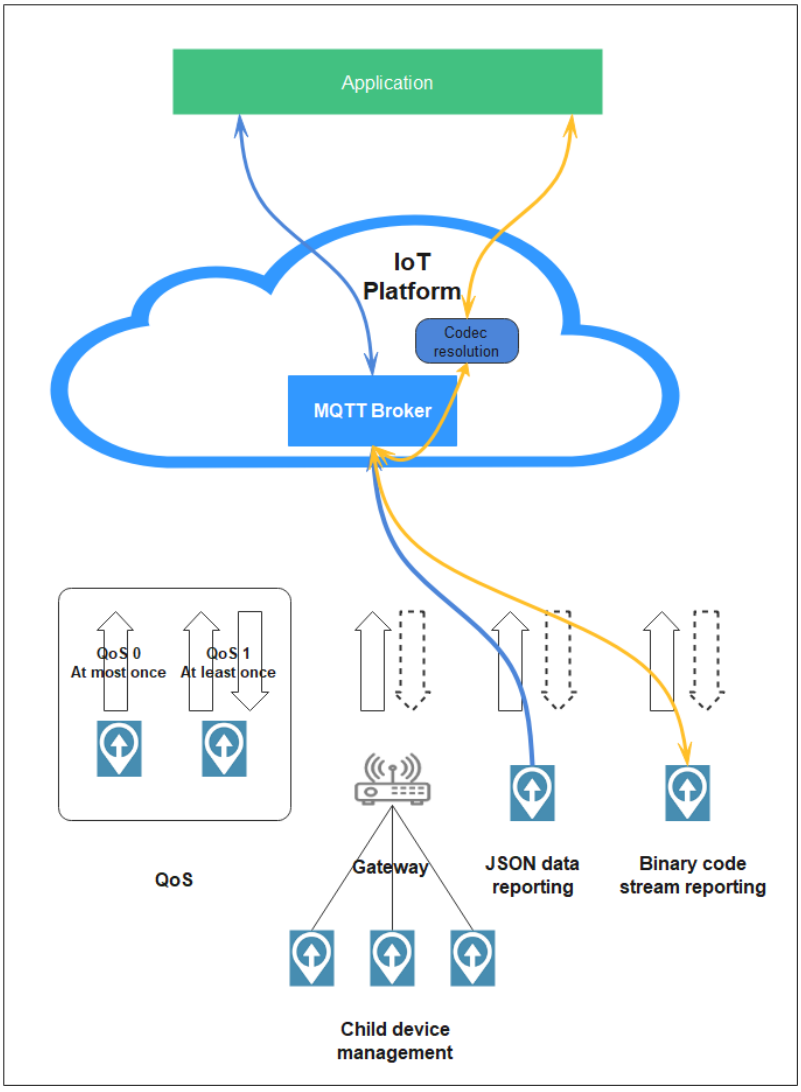
3.5.1 Protocol Introduction

Overview

Message Queuing Telemetry Transport (MQTT) is a publish/subscribe messaging protocol that transports messages between clients and servers. It is suitable for remote sensors and control devices (such as smart street lamps) that have limited

computing capabilities and work in low-bandwidth, unreliable networks through persistent device-cloud connections. MQTT clients publish or subscribe to messages through topics. MQTT brokers centrally manage message routing and ensure end-to-end message transmission reliability based on the preset quality of service (QoS). In this process, the client that sends messages (publisher) and the client that receives messages (subscriber) are decoupled, eliminating the need for a direct connection between them. MQTT has emerged as a top protocol in the IoT domain by meeting the lightweight, reliable, bidirectional, and scalable communication protocol needs of IoT applications. To learn more about the MQTT syntax and interfaces, click **here**.

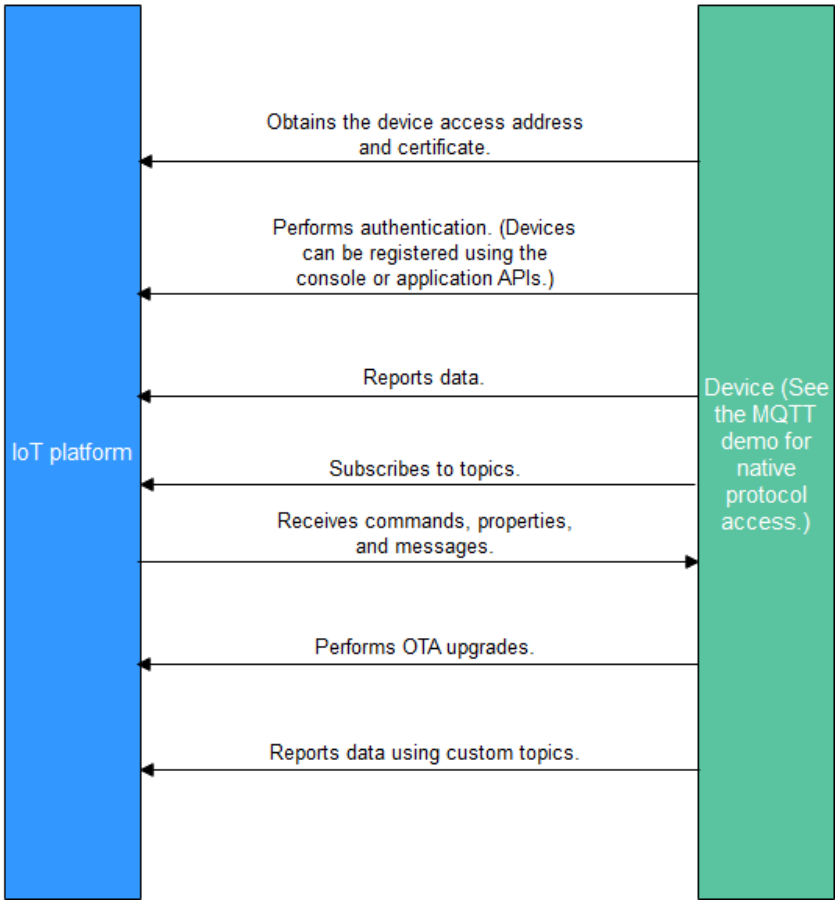
MQTTS is a variant of MQTT that uses TLS encryption. MQTTS devices communicate with the platform using encrypted data transmission.



Service Flow

MQTT devices communicate with the platform without data encryption. For security purposes, MQTTS access is recommended.

You are advised to use the **IoT Device SDK** to connect devices to the platform over MQTTS.



1. Create a product on the IoTDA console or by calling the API **Creating a Product**.
2. Register a device on the **IoTDA console** or calling the API **Creating a Device**.
3. The registered device can report messages and properties, receive commands, properties, and messages, perform OTA upgrades, and report data using custom topics. For details about preset topics of the platform, see **Topic Definition**.

NOTE

You can use MQTT.fx to debug access using the native MQTT protocol. For details, see **Developing an MQTT-based Smart Street Light Online**.

Constraints

Description	Constraint
Number of concurrent connections to a directly connected MQTT device	1
Connection setup requests of an account per second on the device side	• Basic edition: 100 • Standard edition: See **Specifications**.

Description	Constraint
Number of upstream requests for an instance per second on the device side (when average message payload is 512 bytes)	• Basic edition: 500 • Standard edition: See **Specifications**.
Number of upstream messages for an MQTT connection	50 per second
Bandwidth of an MQTT connection (upstream messages)	1 MB (default)
Length of a publish message sent over an MQTT connection (Oversized messages will be rejected.)	1 MB
Standard MQTT protocol	MQTT v5.0, MQTT v3.1.1, and MQTT v3.1
Differences from the standard MQTT protocol	• Not supported: QoS 2 • Not supported: **will** and **retain msg**
Security levels supported by MQTT	TCP channel and TLS protocols (TLS v1, TLS v1.1, TLS v1.2, and TLS v1.3)
Recommended heartbeat interval for MQTT connections	Range: 30s to 1200s; recommended: 120s
MQTT message publish and subscription	A device can only publish and subscribe to messages of its own topics.
Number of subscriptions for an MQTT connection	100
Length of a custom MQTT topic	128 bytes
Number of custom MQTT topics added to a product	10
Number of CA certificates uploaded for an account on the device side	100

Communication Between MQTT Devices and the Platform

The platform communicates with MQTT devices through topics, and they exchange messages, properties, and commands using preset topics. You can also create custom topics for connected devices to meet specific requirements.

Data Type	Message Type	Description
Upstream data	Reporting device properties	Devices report property data in the format defined in the product model.
	Reporting device messages	If a device cannot report data in the format defined in the product model, the device can report data to the platform using the device message reporting API. The platform forwards the messages reported by devices to an application or other Huawei Cloud services for storage and processing.
	Gateway reporting device properties in batches	A gateway reports property data of multiple devices to the platform.
	Reporting device events	Devices report event data in the format defined in the product model.
Downstream data	Delivering platform messages	The platform delivers data in a custom format to devices.
	Setting device properties	A product model defines the properties that the platform can configure for devices. The platform or application can modify the properties of a specific device.
	Querying device properties	The platform or application can query real-time property data of a specific device.
	Delivering platform commands	The platform or application delivers commands in the format defined in the product model to devices.
	Delivering platform events	The platform or application delivers events in the format defined in the product model to devices.

Preset Topics

The following table lists the preset topics of the platform.

Category	Function	Topic	Publisher	Subscriber
Device message related topics	Device Reporting a Message	$oc/devices/{device_id}/sys/messages/up	Device	Platform
	Platform Delivering a Message	$oc/devices/{device_id}/sys/messages/down	Platform	Device
Device command related topics	Platform Delivering a Command	$oc/devices/{device_id}/sys/commands/request_id={request_id}	Platform	Device
	Device Returning a Command Response	$oc/devices/{device_id}/sys/commands/response/request_id={request_id}	Device	Platform
Device property related topics	Device Reporting Properties	$oc/devices/{device_id}/sys/properties/report	Device	Platform
	Reporting Property Data by a Gateway	$oc/devices/{device_id}/sys/gateway/sub_devices/properties/report	Device	Platform
	Setting Device Properties	$oc/devices/{device_id}/sys/properties/set/request_id={request_id}	Platform	Device
	Returning a Response to Property Settings	$oc/devices/{device_id}/sys/properties/set/response/request_id={request_id}	Device	Platform
	Querying Device Properties	$oc/devices/{device_id}/sys/properties/get/request_id={request_id}	Platform	Device

Category	Function	Topic	Publisher	Subscriber
	Device Returning a Response for a Property Query The response does not affect device properties and shadows.	$oc/devices/{device_id}/sys/properties/get/response/request_id={request_id}	Device	Platform
	Obtaining Device Shadow Data from the Platform	$oc/devices/{device_id}/sys/shadow/get/request_id={request_id}	Device	Platform
	Returning a Response to a Request for Obtaining Device Shadow Data	$oc/devices/{device_id}/sys/shadow/get/response/request_id={request_id}	Platform	Device
Device event related topics	**Reporting a Device Event**	$oc/devices/{device_id}/sys/events/up	Device	Platform
	Delivering an Event	$oc/devices/{device_id}/sys/events/down	Platform	Device

You can create custom topics on the console to report personalized data. For details, see **Custom Topic Communications**.

TLS Support for MQTT

TLS is recommended for secure transmission between devices and the platform. Currently, TLS v1.1, v1.2, v1.3, and GMTLS are supported. TLS v1.3 is recommended. TLS v1.1 will not be supported in the future. GMTLS is supported only by the enterprise edition using Chinese cryptographic algorithms.

When TLS connections are used for the basic edition, standard edition, and enterprise edition that support general cryptographic algorithms, the IoT platform supports the following cipher suites:

- TLS_AES_256_GCM_SHA384
- TLS_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA

When the enterprise edition that supports Chinese cryptographic algorithms uses TLS connections, the IoT platform supports the following cipher suites:

- ECC_SM4_GCM_SM3
- ECC_SM4_CBC_SM3
- ECDHE_SM4_GCM_SM3
- ECDHE_SM4_CBC_SM3
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

NOTE

CBC cipher suites may pose security risks.

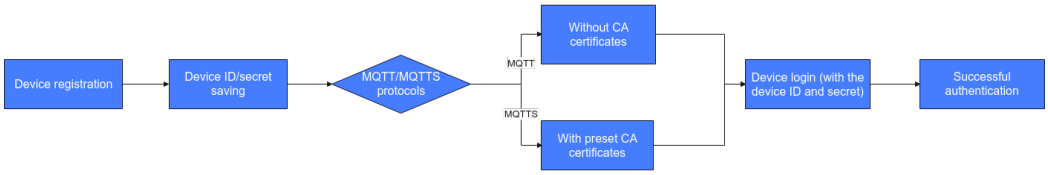
3.5.2 Secret Authentication

Overview

MQTT(S) secret authentication requires a device to have its ID and secret for access authentication. For devices connected through MQTTS, a CA certificate must be preconfigured on the devices.

Process

Figure 3-159 MQTT(S) secret authentication process



1. An application calls the API for registering a device. Alternatively, a user uses the IoTDA console to register a device.

   NOTE

   During registration, use the MAC address, serial number, or IMEI of the device as the node ID.

2. The platform allocates a globally unique device ID and secret to the device.

> **NOTE**
>
> The secret can be defined during device registration. If no secret is defined, the platform allocates one.

3. The device needs to integrate **the preset CA certificate** (only for the authentication process of MQTTS access).
4. During login, the device sends a connection request carrying the device ID and secret.
5. If the authentication is successful, the platform returns a success message, and the device is connected to the platform.

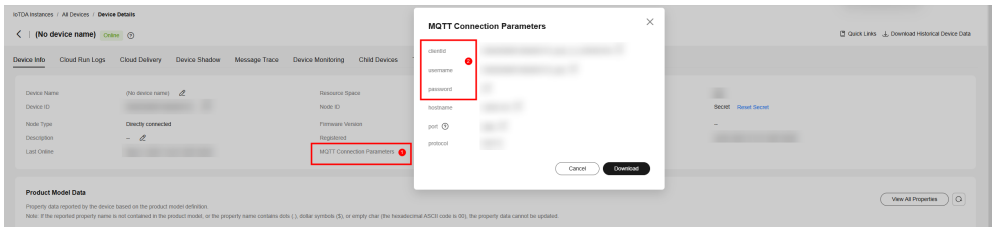
Procedure

This section uses MQTT.fx to describe how to activate a device registered on the IoT platform.

Step 1 Download **MQTT.fx** (64-bit OS) or **MQTT.fx** (32-bit OS) and install it.

Step 2 Go to the device details page, click **MQTT Connection Parameters**, and check the device connection information (**ClientId**, **Username**, and **Password**).

Figure 3-160 Device - Connection parameters



Alternatively, access the **parameter generation tool** and enter the device ID (**device_id**) and secret (**secret**) generated after registration to generate the parameters (**ClientId**, **Username**, and **Password**) required for device connection authentication.

Table 3-27 Parameters

Parameter	Mandatory	Type	Description
ClientId	Yes	String	**Definition** The value of this parameter consists of a device ID, device type, password signature type, and timestamp. They are separated by underscores (_). ● Device ID: A device ID uniquely identifies a device and is generated when the device is registered with IoTDA. The value usually consists of a device's product ID and node ID which are separated by an underscore (_). ● Device type: The value is fixed at **0**, indicating a device ID. ● Password signature type: The length is 1 byte, and the value can be **0** or **1**. – **0**: The timestamp is not verified using the HMAC-SHA256 algorithm. – **1**: The timestamp is verified using the HMAC-SHA256 algorithm. ● Timestamp: The UTC time when the device was connected to IoTDA. The format is YYYYMMDDHH. For example, if the UTC time is 2018/7/24 17:56:20, the timestamp is **2018072417**. **Range** Up to 256 characters.
UserName	Yes	String	**Definition** The value is the device ID (**device_id**). **Range** Up to 256 characters.

Parameter	Mandatory	Type	Description
Password	Yes	String	**Definition** With the timestamp (in YYYYMMDDHH format) as the key, use the HMAC-SHA256 algorithm to encrypt the device secret returned by IoTDA upon successful device registration. The result is the password. Password = hmacsha256 ("secret", "timestamp") Set this parameter only if the device authentication type is **SECRET**. Not required for X.509 certificate authentication (**CERTIFICATES**). HMACSHA256 is an HMAC algorithm that uses SHA-256 to generate a hash value. The generated hash value is represented by a 64-bit hexadecimal string. For example, if the device secret is **12345678** and the timestamp is **2025041401**, the result is **c75150e6cb841417396819e4d2ee4358a416344a03a083e3a8567074ddec820a**. **Range** Up to 256 characters.

Each device performs authentication using the MQTT CONNECT message, which must contain all information of the client ID. After receiving a CONNECT message, the platform checks the authentication type and password digest algorithm of the device.

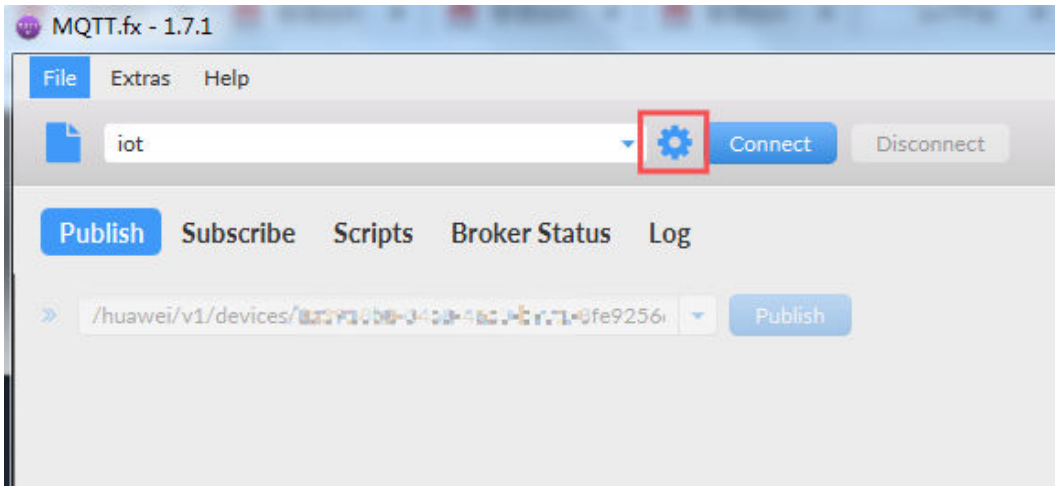
The generated client ID is in the format ***Device ID*_0_0_*Timestamp***. By default, the timestamp is not verified.

- If the timestamp needs to be verified, the platform checks whether the message timestamp is consistent with the platform time and then checks whether the password is correct.
- If the timestamp does not need to be verified, the timestamp must also be contained in the CONNECT message, but the platform does not check whether the time is correct. In this case, only the password is checked.

If the authentication fails, the platform returns an error message and automatically disconnects the MQTT connections.

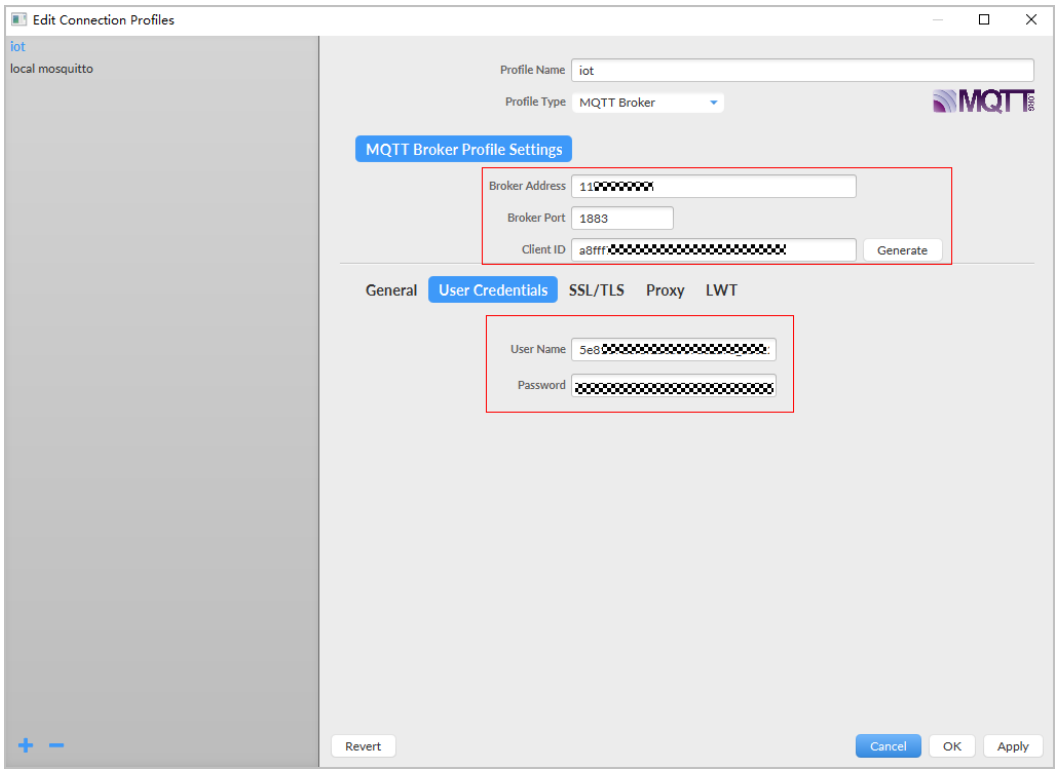
Step 3 Open MQTT.fx and click the setting icon.

Figure 3-161 MQTT.fx - Setting



Step 4 Configure authentication parameters and click **Apply**.

Figure 3-162 Connection configuration

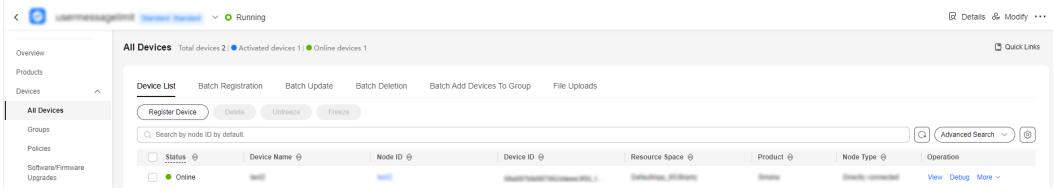


Parameter	Description
Broker Address	Enter the **device access address** (domain name) obtained from the IoTDA console. For devices that cannot be connected to the platform using a domain name, run the **ping** *Domain name* command in the CLI to obtain the IP address. The IP address is variable and needs to be set using a configuration item.

Parameter	Description
Broker Port	For MQTT non-encrypted protocols, the port number is 1883, which is the default value. For MQTTS encrypted protocols, change the port number to 8883 and **obtain the certificate** for verifying the IoT platform identity. For details, see **Using MQTT.fx to Simulate Communication Between the Smart Street Light and the Platform**.
Client ID	Enter the device client ID obtained in **2**.
User Name	Enter the device ID obtained in **2**.
Password	Enter the encrypted device secret obtained in **2**.

Step 5 Click **Connect**. If the device authentication is successful, the device is displayed online on the platform.

Figure 3-163 Device online status



----End

Best Practices

Developing an MQTT-based Simulated Smart Street Light Online

3.5.3 Certificate Authentication

3.5.3.1 Usage

Introduction

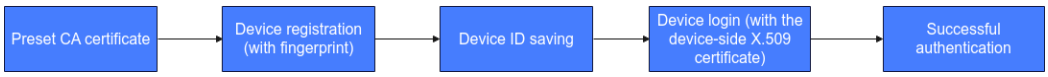
MQTT(S) certificate authentication requires you to upload a device Certificate Authority (CA) certificate on the console first. Then, you can either use the API for **creating a device** or register the device on the console to get the device ID. When the device accesses the IoT platform, it carries the X.509 certificate for authentication, which is a digital certificate used to authenticate the communication entity.

Constraints

- Only MQTT(S) devices can use X.509 certificates for identity authentication.
- You can upload up to 100 device CA certificates. Multiple devices can share one CA certificate.

Process

Figure 3-164 Process



1. A user uploads a device CA certificate on the IoTDA console.
2. An application calls the API for registering a device. Alternatively, a user uses the IoTDA console to register a device.

   NOTE

   During registration, use the MAC address, serial number, or IMEI of the device as the node ID.

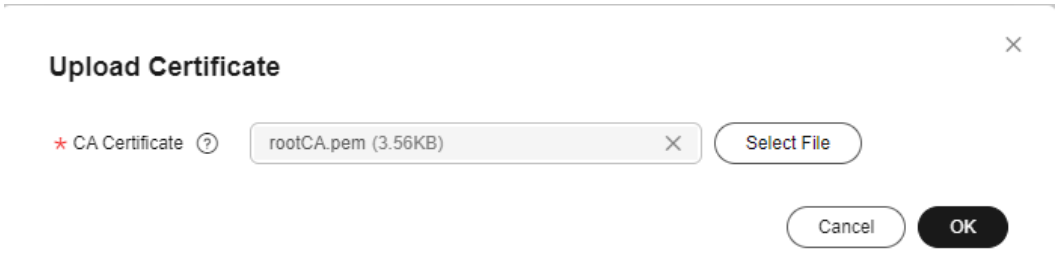
   The platform allocates a globally unique device ID to the device.
3. During login, the device sends a connection request carrying the **X.509 certificate** to the platform.
4. If the authentication is successful, the platform returns a success message, and the device is connected to the platform.

Uploading a Device CA certificate

Step 1 In the navigation pane, choose **Devices** > **Device Certificates**. On the **Device CA Certificates** tab page, select a resource space and click **Upload Certificate**.

Step 2 In the displayed dialog box, click **Select File** to add a file, and then click **OK**.

Figure 3-165 Device CA certificate - Uploading a certificate



NOTE

- Device CA certificates are provided by device vendors. You can **prepare a commissioning certificate** during commissioning. For security reasons, you are advised to replace the commissioning certificate with a commercial certificate during commercial use.
- CA certificates can no longer be used to verify server certificates upon expiration. Replace CA certificates before they expire to ensure that devices can connect to the IoT platform properly.

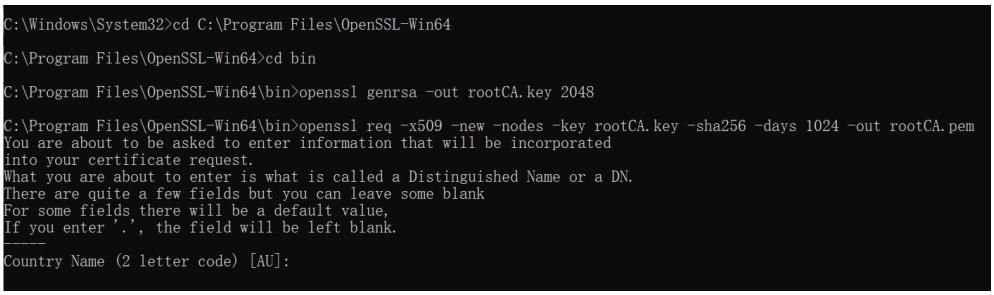
----End

Creating a Device CA Commissioning Certificate

This section uses the Windows operating system as an example to describe how to use OpenSSL to make a commissioning certificate. The generated certificate is in PEM format.

1. Download and install **OpenSSL**.
2. Open the CLI as user **admin**.
3. Run **cd c:\openssl\bin** (replace **c:\openssl\bin** with the actual OpenSSL installation directory) to access the OpenSSL view.
4. Generate a public/private key pair.
   ```
   openssl genrsa -out rootCA.key 2048
   ```
5. Use the private key in the key pair to generate a CA certificate.
   ```
   openssl req -x509 -new -nodes -key rootCA.key -sha256 -days 1024 -out rootCA.pem
   ```

   Figure 3-166 Generating a CA certificate

   Enter the following information as prompted. All parameters can be customized.

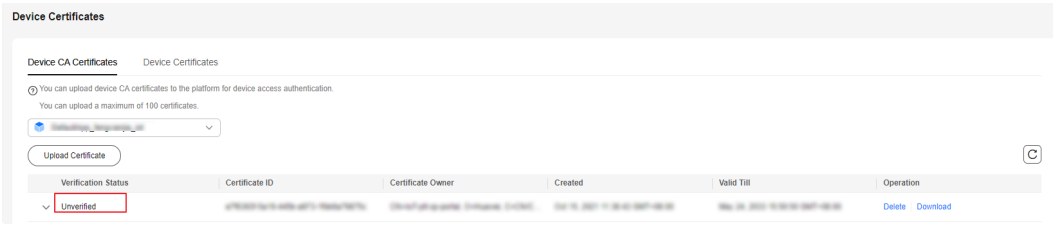
   - Country Name (2 letter code) [AU]: country, for example, CN
   - State or Province Name (full name) []: state or province, for example, GD
   - Locality Name (for example, city) []: city, for example, SZ
   - Organization Name (for example, company) []: organization, for example, Huawei
   - Organizational Unit Name (for example, section) []: organization unit, for example, IoT
   - Common Name (e.g. server FQDN or YOUR name) []: common name, for example, zhangsan
   - Email Address []: email address, for example, 1234567@163.com

   Obtain the generated CA certificate **rootCA.pem** from the **bin** folder in the OpenSSL installation directory.

Uploading a Verification Certificate

If the uploaded certificate is a commissioning certificate, the certificate status is **Unverified**. In this case, upload a verification certificate to verify that you have the CA certificate.

Figure 3-167 Device CA certificate - Unverified certificate



The verification certificate is created based on the private key of the device CA certificate. Perform the following operations to create a verification certificate:

Step 1 Obtain the verification code to verify the certificate.

Figure 3-168 Device CA certificate - Verifying a certificate

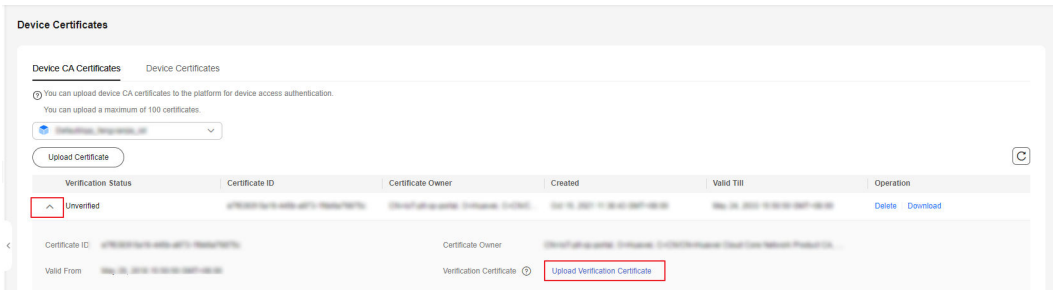
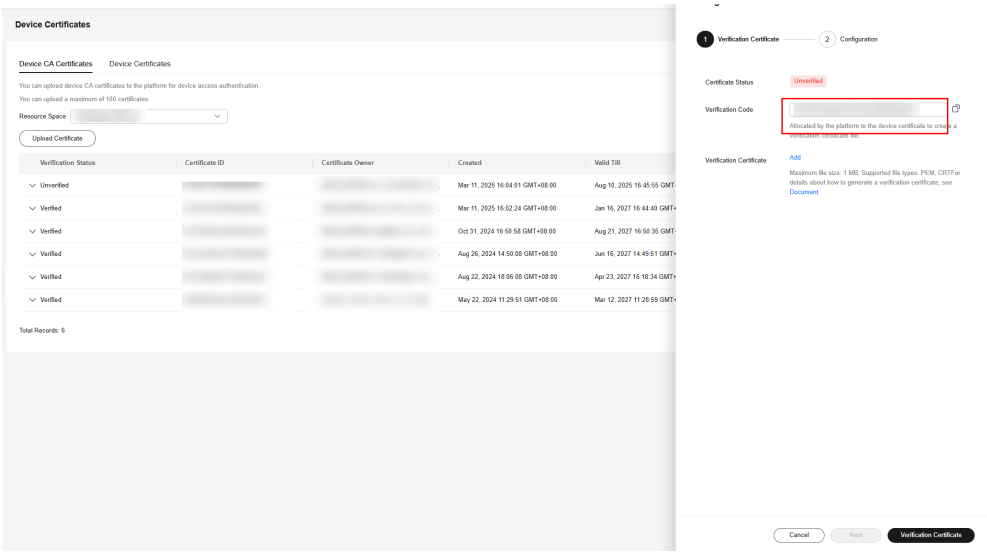


Figure 3-169 Device CA certificate - Obtaining the verification code



Step 2 Generate a key pair for the verification certificate.

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

Step 3 Create a certificate signing request (CSR) for the verification certificate.

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

The system prompts you to enter the following information. Set **Common Name** to the verification code and set other parameters as required.

- Country Name (2 letter code) [AU]: country, for example, CN

- State or Province Name (full name) []: state or province, for example, GD
- Locality Name (for example, city) []: city, for example, SZ
- Organization Name (for example, company) []: organization, for example, Huawei
- Organizational Unit Name (for example, section) []: organization unit, for example, IoT
- Common Name (e.g. server FQDN or YOUR name) []: verification code for verifying the certificate. For details on how to obtain the verification code, see **Step 1**.
- Email Address []: email address, for example, 1234567@163.com
- Password[]: password
- Optional Company Name[]: company name, for example, Huawei

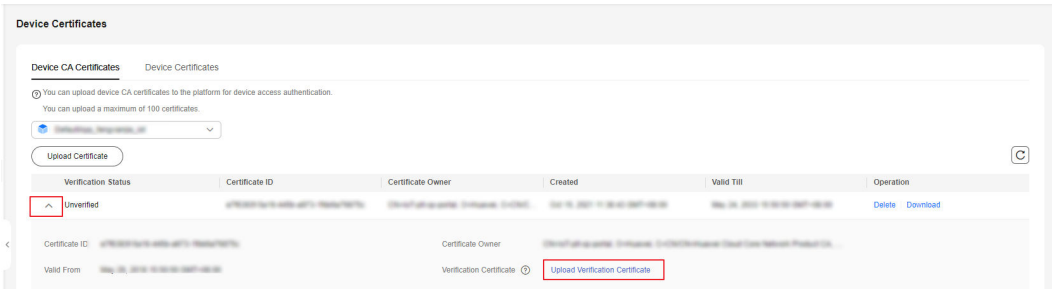
Step 4 Use the CSR to create a verification certificate.

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

Obtain the generated verification certificate **verificationCert.pem** from the **bin** folder of the OpenSSL installation directory.

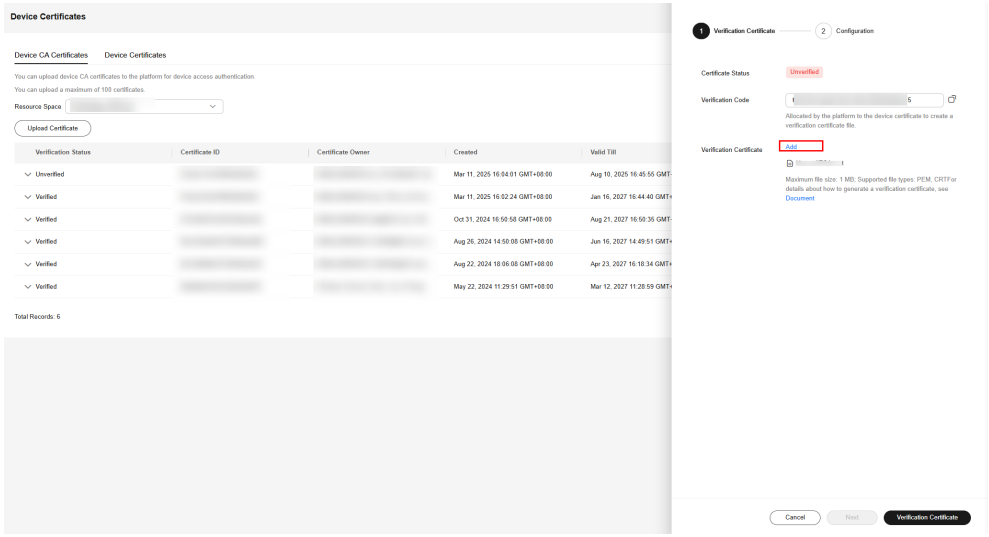
Step 5 Select the corresponding certificate, click , and click **Upload Verification Certificate**.

Figure 3-170 Device CA certificate - Verifying a certificate



Step 6 In the displayed dialog box, click **Select File** to add a file, and then click **OK**.

Figure 3-171 Device CA certificate - Uploading a verified certificate



After the verification certificate is uploaded, the certificate status changes to **Verified**, indicating that you have the CA certificate.

----End

Presetting an X.509 Certificate

Before registering an X.509 device, preset the X.509 certificate issued by the CA on the device.

NOTE

The X.509 certificate is issued by the CA. If no commercial certificate issued by the CA is available, you can **create a device CA commissioning certificate**.

Creating an X.509 Commissioning Certificate

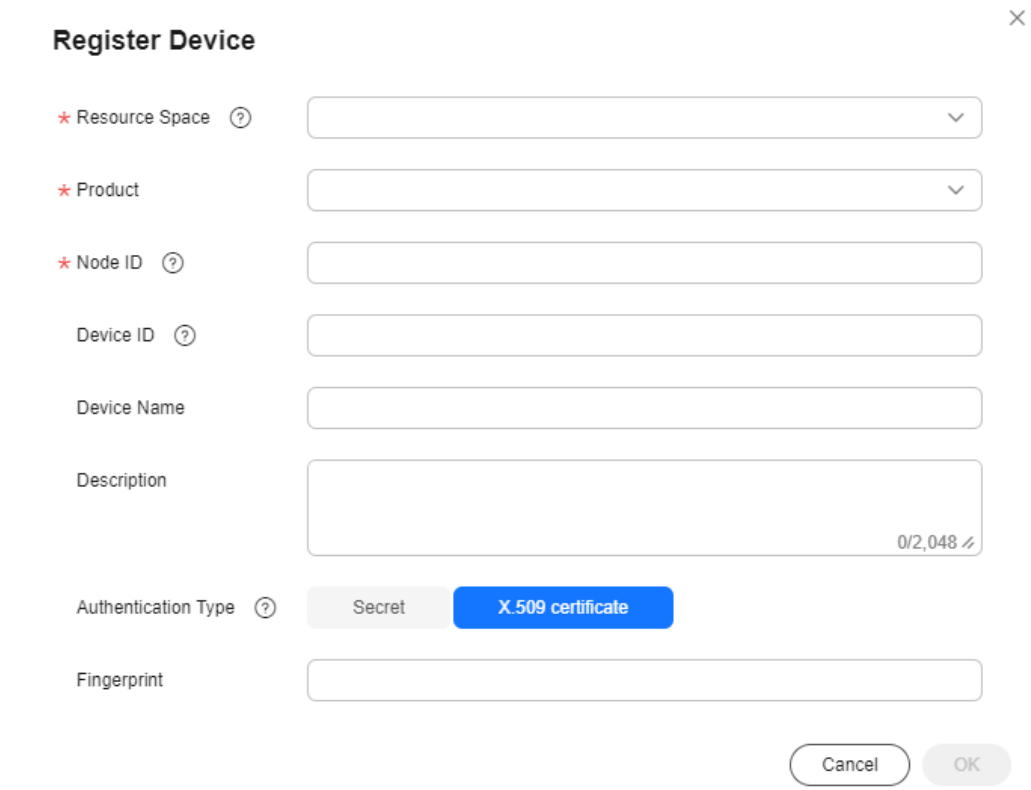
1. Run **cmd** as user **admin** to open the CLI and run **cd c:\openssl\bin** (replace **c:\openssl\bin** with the actual OpenSSL installation directory) to access the OpenSSL view.
2. Generate a public/private key pair.
   ```
   openssl genrsa -out deviceCert.key 2048
   ```
3. Create a CSR for the device certificate.
   ```
   openssl req -new -key deviceCert.key -out deviceCert.csr
   ```
   Enter the following information as prompted. All parameters can be customized.
   - Country Name (2 letter code) [AU]: country, for example, CN
   - State or Province Name (full name) []: state or province, for example, GD
   - Locality Name (for example, city) []: city, for example, SZ
   - Organization Name (for example, company) []: organization, for example, Huawei
   - Organizational Unit Name (for example, section) []: organization unit, for example, IoT
   - Common Name (e.g. server FQDN or YOUR name) []: common name, for example, zhangsan
   - Email Address []: email address, for example, 1234567@163.com
   - Password[]: password
   - Optional Company Name[]: company name, for example, Huawei
4. Create a device certificate using CSR.
   ```
   openssl x509 -req -in deviceCert.csr -CA rootCA.pem -CAkey rootCA.key -CAcreateserial -out deviceCert.pem -days 500 -sha256
   ```
   Obtain the generated device certificate **deviceCert.pem** from the **bin** folder in the OpenSSL installation directory.

Registering a Device Authenticated by an X.509 Certificate

Step 1 Log in to the **IoTDA console**.

Step 2 In the navigation pane, choose **Devices** > **All Devices**. On the displayed page, click **Register Device**, set parameters based on the table below, and click **OK**.

Figure 3-172 Device - Registering an X.509 device



Parameter	Description
Resource Space	Select the resource space to which a device belongs.
Product	Select the product to which the device belongs.
Node ID	Set this parameter to the IMEI, MAC address, or serial number of the device. If the device is not a physical one, set this parameter to a custom character string that contains letters and digits.
Device Name	Customize the device name.
Authentication Type	**X.509 certificate**: The device uses an X.509 certificate for identity verification.

Parameter	Description

Fingerprint	This parameter is displayed when **Authentication Type** is set to **X.509 certificate**. Import the fingerprint corresponding to the **preset device certificate on the device side**. You can run **openssl x509 -fingerprint -sha256 -in deviceCert.pem** in the OpenSSL view to query the fingerprint. **Note: Delete the colons (:) from the obtained fingerprint when filling it.**
	
	Figure 3-173 OpenSSL execution example
	`[root@k8s-iot-wl2-2-jump cert1223]# openssl x509 -fingerprint -sha256 -in deviceCert.pem`
	`SHA256 Fingerprint=F7:91:90:45:BB:88:37:E6:A7:E7:70:4A:90:75:F3:87:DA:27:5B:7C:49:3E:FF:59:7A:6F:4F:08:4D:F8:54:E8`

----End

Performing Connection Authentication

You can activate the device registered with the platform by using MQTT.fx. For details, see **Device Connection Authentication**.

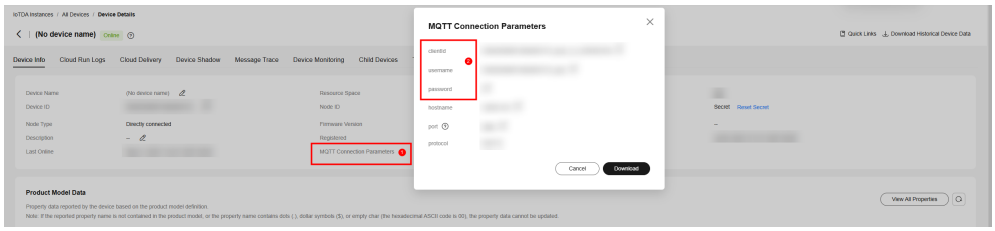
Step 1 Download **MQTT.fx** (64-bit OS) or **MQTT.fx** (32-bit OS) and install it.

NOTE

- Install the latest **MQTT.fx**.
- MQTT.fx 1.7.0 and earlier versions have problems in processing topics containing $. Use the latest version for test.

Step 2 Go to the device details page, click **MQTT Connection Parameters**, and check the device connection information (**ClientId**, **Username**, and **Password**).

Figure 3-174 Device - Connection parameters



Parameter	Mandatory	Type	Description
ClientId	Yes	String(256)	The value of this parameter consists of a device ID, device type, password signature type, and timestamp. They are separated by underscores (_). ● Device ID: A device ID uniquely identifies a device and is generated when the device is registered with IoTDA. The value usually consists of a device's product ID and node ID which are separated by an underscore (_). ● Device type: The value is fixed at **0**, indicating a device ID. ● Password signature type: The length is 1 byte, and the value can be **0** or **1**. – **0**: The timestamp is not verified using the HMAC-SHA256 algorithm. – **1**: The timestamp is verified using the HMAC-SHA256 algorithm. ● Timestamp: The UTC time when the device was connected to IoTDA. The format is YYYYMMDDHH. For example, if the UTC time is 2018/7/24 17:56:20, the timestamp is **2018072417**.
Username	Yes	String(256)	Device ID.

Each device performs authentication using the MQTT CONNECT message, which must contain all information of the client ID. After receiving a CONNECT message, the platform checks the authentication type and password digest algorithm of the device.

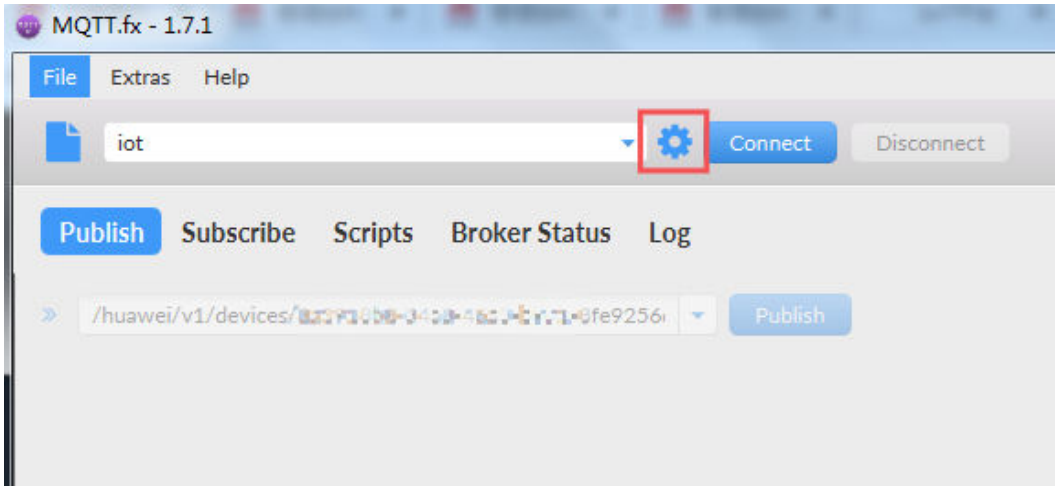
The generated client ID is in the format ***Device ID*_0_0_*Timestamp***. By default, the timestamp is not verified.

- If the timestamp needs to be verified, the platform checks whether the message timestamp is consistent with the platform time and then checks whether the password is correct.
- If the timestamp does not need to be verified, the timestamp must also be contained in the CONNECT message, but the platform does not check whether the time is correct. In this case, only the password is checked.

If the authentication fails, the platform returns an error message and automatically disconnects the MQTT connections.

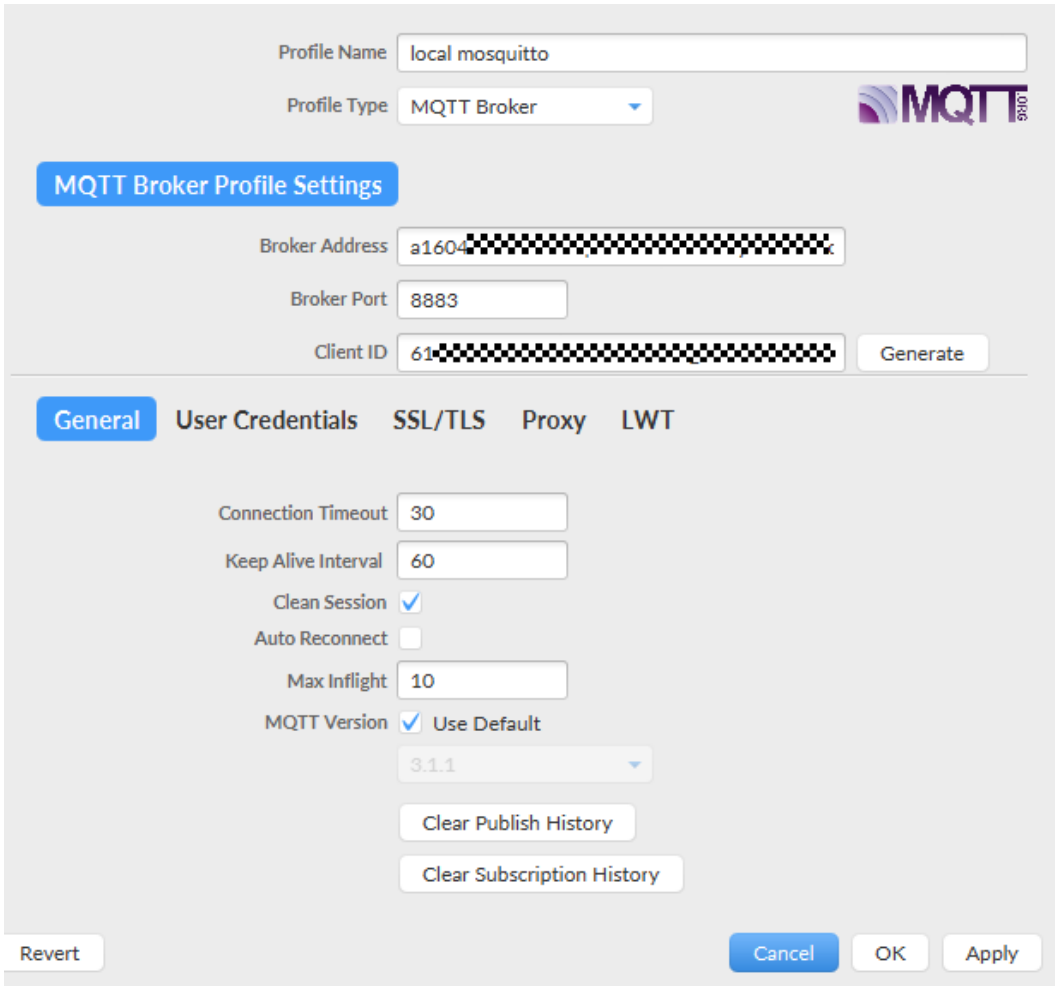
Step 3 Open MQTT.fx and click the setting icon.

Figure 3-175 MQTT.fx - Settings



Step 4 Enter **Connection Profile** information.

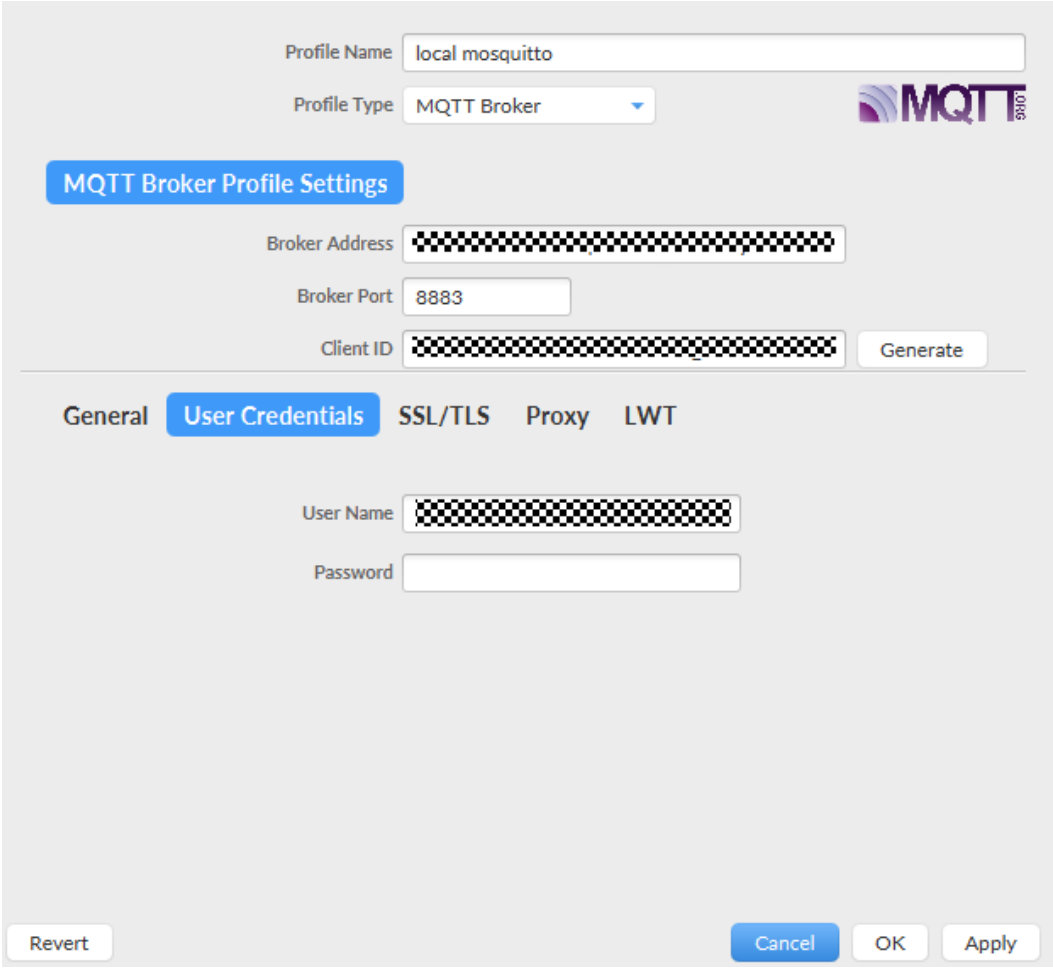
Figure 3-176 Using default settings for parameters on the General tab page



Parameter	Description
Broker Address	Enter the **device access address** (domain name) obtained from the IoTDA console. For devices that cannot be connected to the platform using a domain name, run the **ping** *Domain name* command in the CLI to obtain the IP address. The IP address is variable and needs to be set using a configuration item.
Broker Port	Enter **8883**.
Client ID	Enter the device client ID obtained in **2**.

Step 5 Click **User Credentials** and specify **User Name**.

Figure 3-177 Entering the device ID

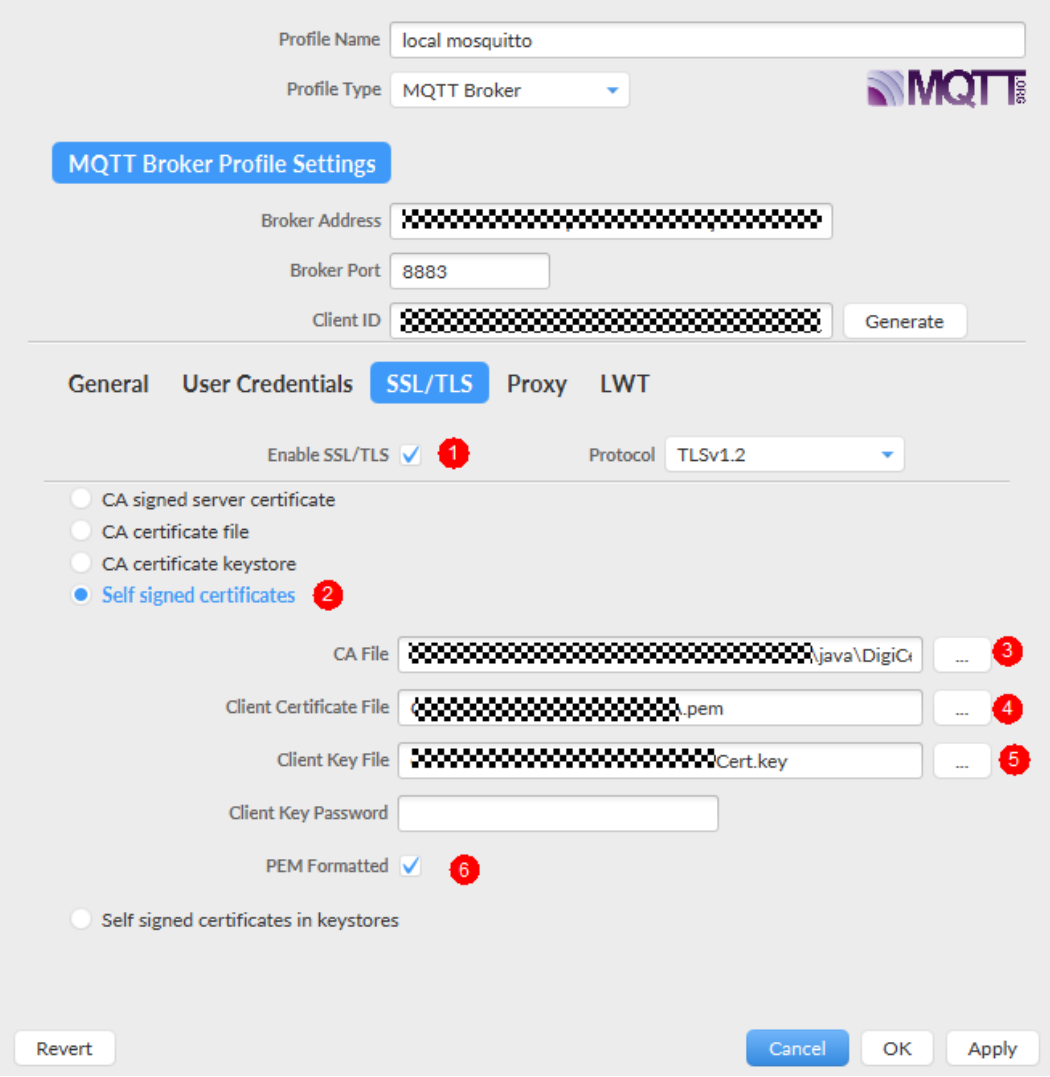


Parameter	Description
User Name	Enter the device ID obtained in **2**.

Parameter	Description
Password	Leave it blank when the X.509 certificate is used for authentication.

Step 6 Click **SSL/TLS**, set authentication parameters, and click **Apply**. Select **Enable SSL/TLS**, select **Self signed certificates**, and enter the certificate information.

Figure 3-178 Setting SSL/TLS parameters

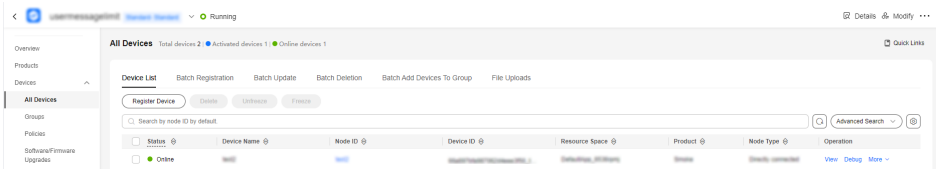


NOTE

- **CA File**: corresponding CA certificate. Download the certificate from **Obtaining Resources** and load the PEM certificate.
- **Client Certificate File**: device certificate (deviceCert.pem).
- **Client Key File**: private key (deviceCert.key) of the device.

Step 7 Click **Connect**. If the device authentication is successful, the device is displayed online on the platform.

Figure 3-179 Device list - Device online status



----End

APIs

- **Create a Device**
- **Reset a Device Secret**
- **Obtain the Device CA Certificate List**
- **Upload a Device CA Certificate**
- **Delete a Device CA Certificate**
- **Verify a Device CA Certificate**

Best Practices

Connecting a Device That Uses the X.509 Certificate Based on MQTT.fx

3.5.3.2 Certificate Validity Verification (OCSP)

Introduction

IoTDA uses Online Certificate Status Protocol (OCSP) to verify the validity of certificates on the device and server. **OCSP** checks the revocation status of certificates at the Transport Layer Security (TLS) layer. It offers several advantages over the traditional Certificate Revocation List (CRL), including higher scalability, shorter response time, better real-time performance, and greater suitability for the Public Key Infrastructure (PKI). Unlike CRL, which is updated less frequently and has a larger file size, OCSP provides more efficient and timely certificate verification.

Terms

OCSP verification: used for device certificate validity status check on the platform side. The IoT platform checks whether the device certificate has been revoked by the CA.

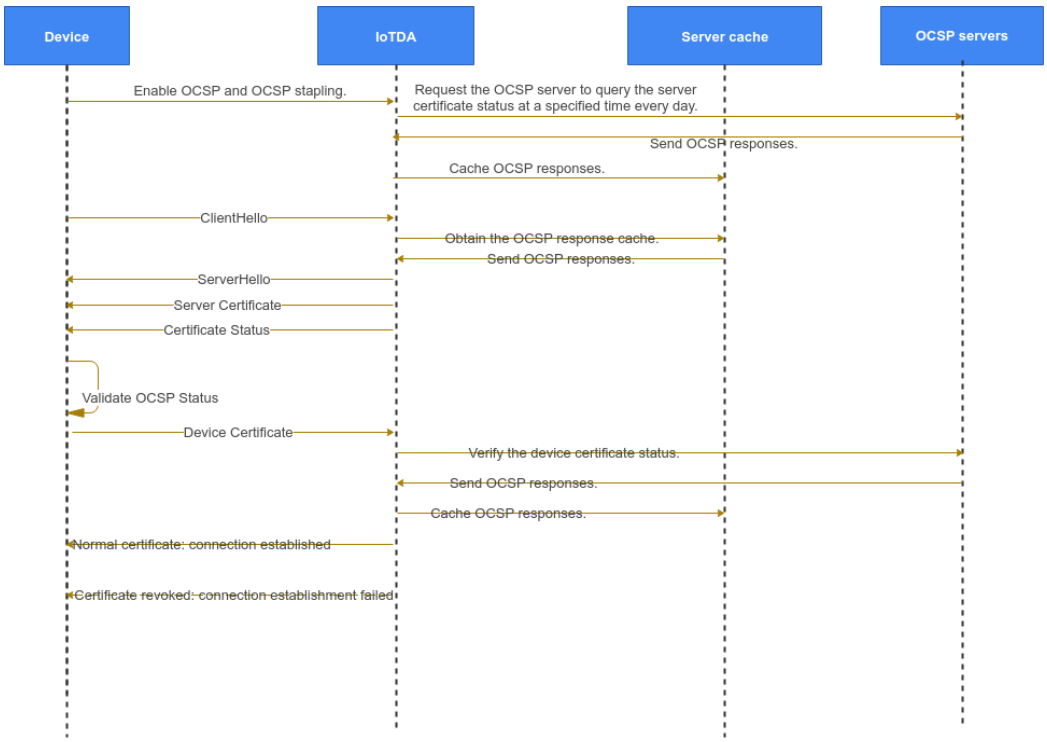
OCSP stapling: also known as server OCSP, is a TLS certificate status query extension that serves as an alternative to traditional OCSP for checking the status of X.509 certificates. With OCSP stapling, the server takes the initiative to check its certificate revocation status (continuously) and includes a cached OCSP response during the TLS handshake. This eliminates the need to send a separate request to the CA and speeds up the handshake process, as you only need to verify the validity of the response.

Constraints

- Only enterprise instances support this function.
- When OCSP verification is enabled, the platform will send a request to the OCSP server during the initial device connection to the platform. This may result in a longer duration for establishing the connection, which is a normal occurrence. Subsequent connection establishment is not affected.
- Cache duration for the platform to respond to the OCSP server: 24 hours.
- Timeout interval for the platform to respond to the OCSP server: 5 seconds; max. response size: 4 KB.

Process

Figure 3-180 OCSP working process

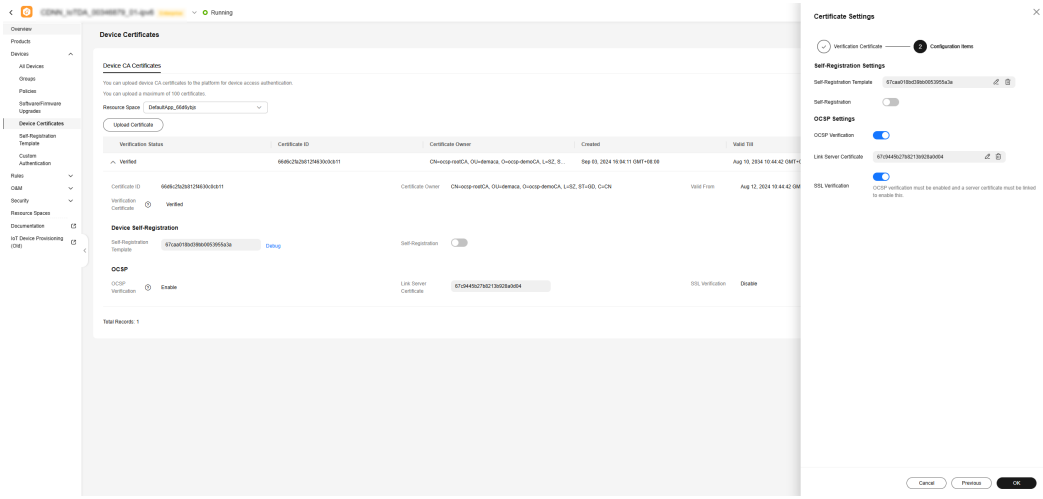


Procedure

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card.

Step 2 In the navigation pane, choose **Devices** > **Device Certificates**. On the displayed page, click the corresponding CA certificate and click the button for certificate settings.

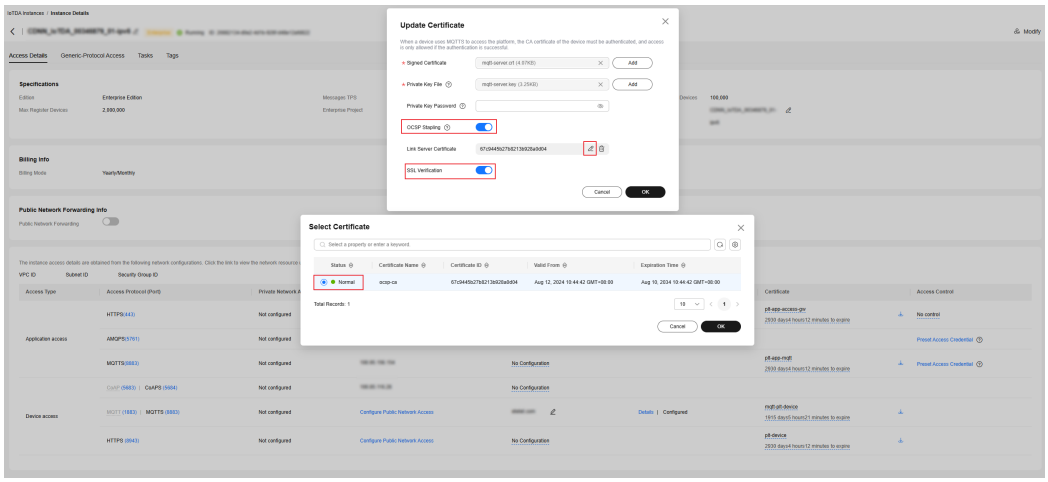
Figure 3-181 Device CA certificate - OCSP verification enabled



Step 3 If the OCSP server accesses the platform using HTTPS, choose **Rules Server Certificates** in the navigation pane, click **Upload Certificate**, and upload the CA certificate of the OCSP server.

Return to the IoTDA console, select the corresponding instance, and access its details page. Click **Update Certificate** to enable OCSP stapling. If the OCSP server accesses the platform using HTTPS, click **SSL Verification** and associate the server certificate.

Figure 3-182 Instance management - Enabling OCSP stapling



NOTE

- To enable OCSP stapling, the certificate chain must contain the upper-layer CA certificate.
- The OCSP signature certificate information must contain the OCSP URL extension field.

Step 4 Use the MQTT simulator that supports OCSP to connect to the platform, check the OCSP stapling information of the platform certificate, use the packet capture tool to capture TLS handshake packets for connection establishment, and check the OCSP response of the platform certificate. There are three certificate status types:

good, revoked, and unknown. The device determines whether to establish a connection based on the platform certificate status. For example, the device establishes a connection with the platform only when the certificate status is good.

Figure 3-183 TLS-Certificate Status good

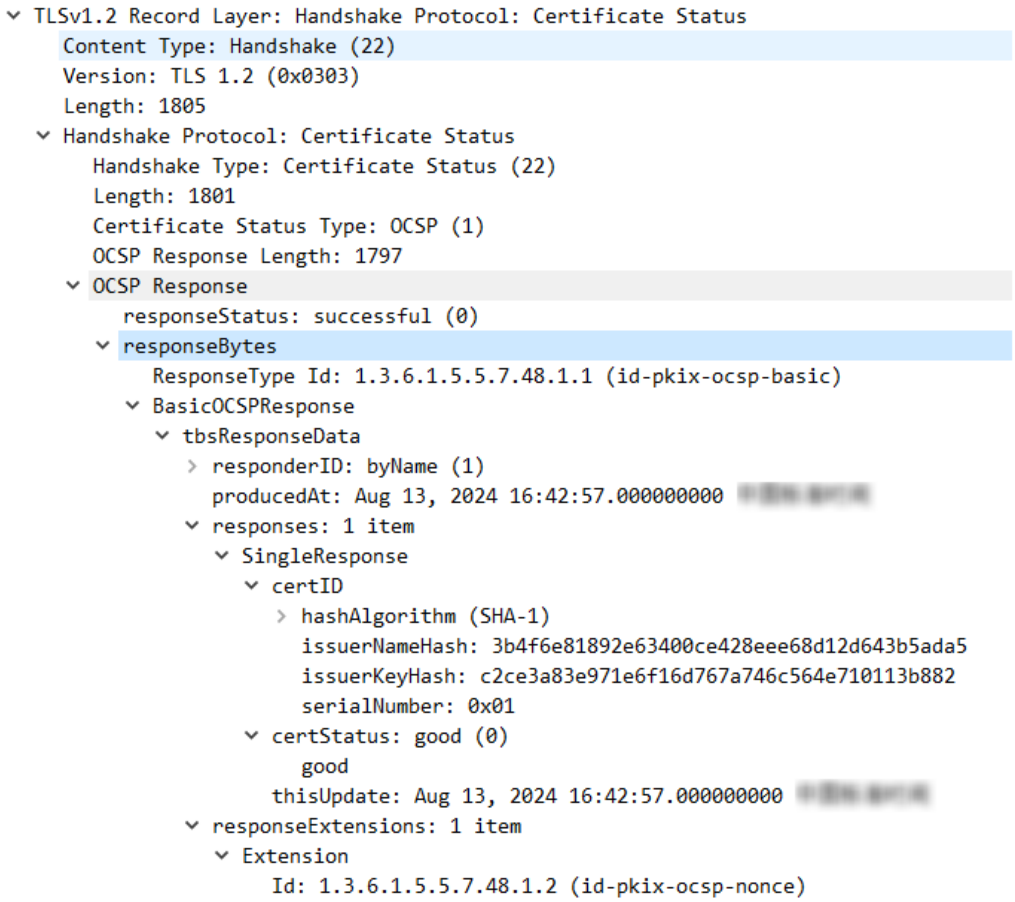


Figure 3-184 TLS-Certificate Status revoked

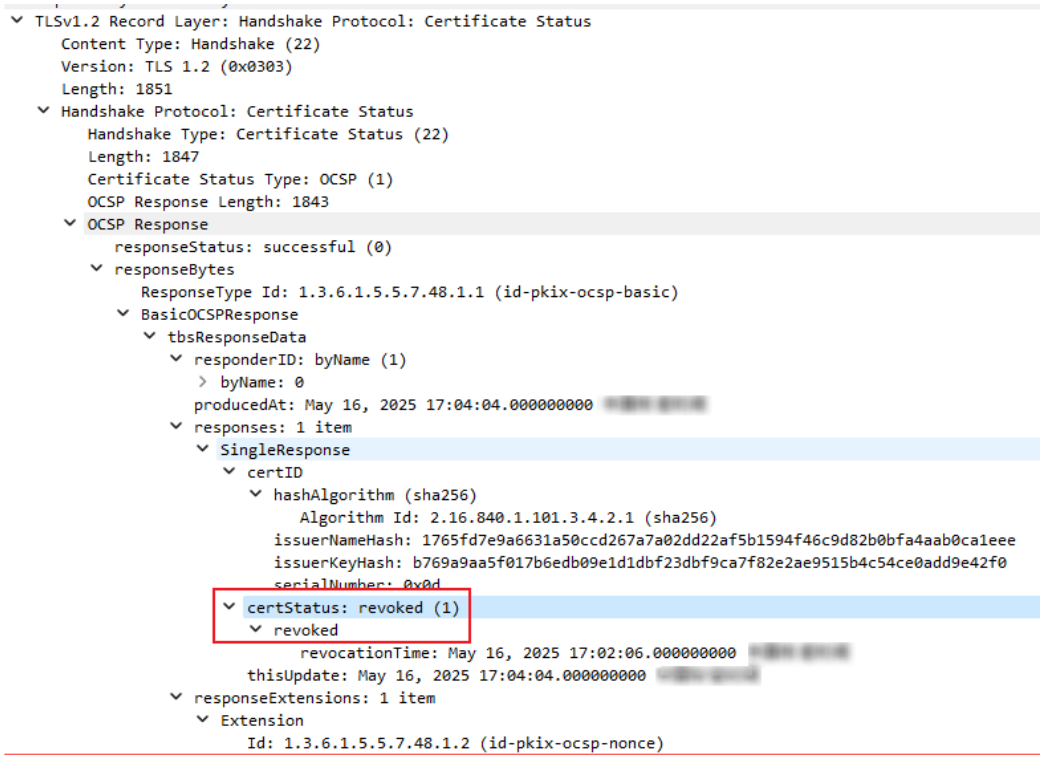
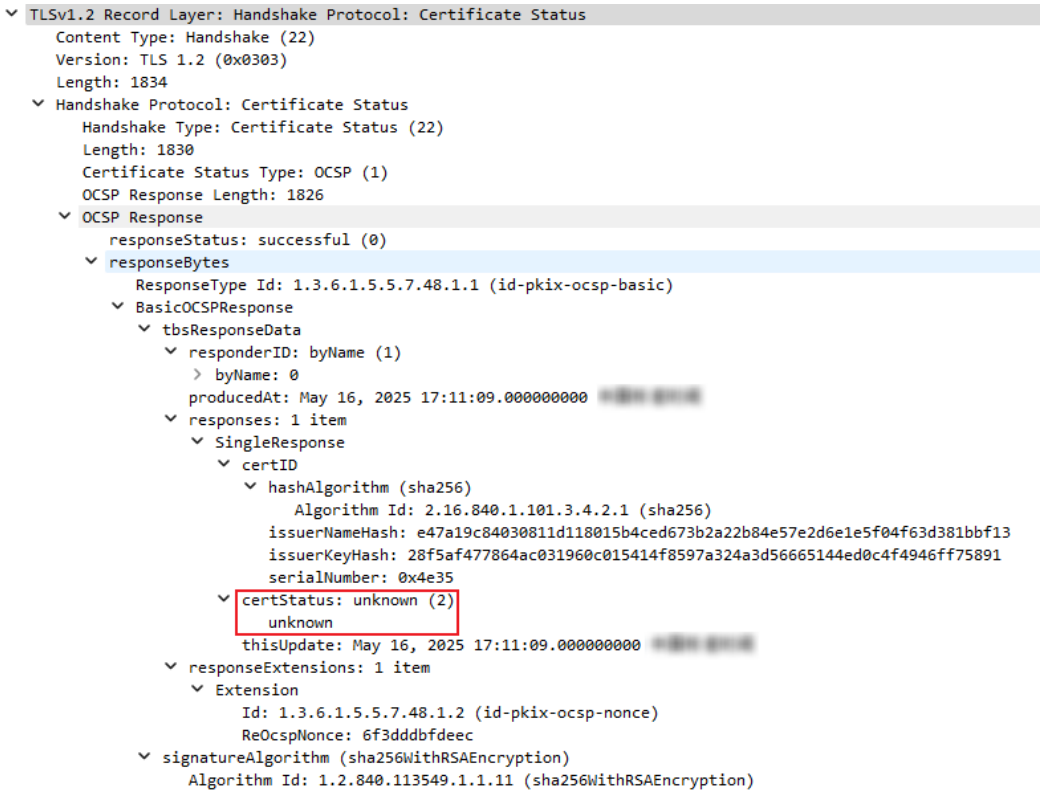


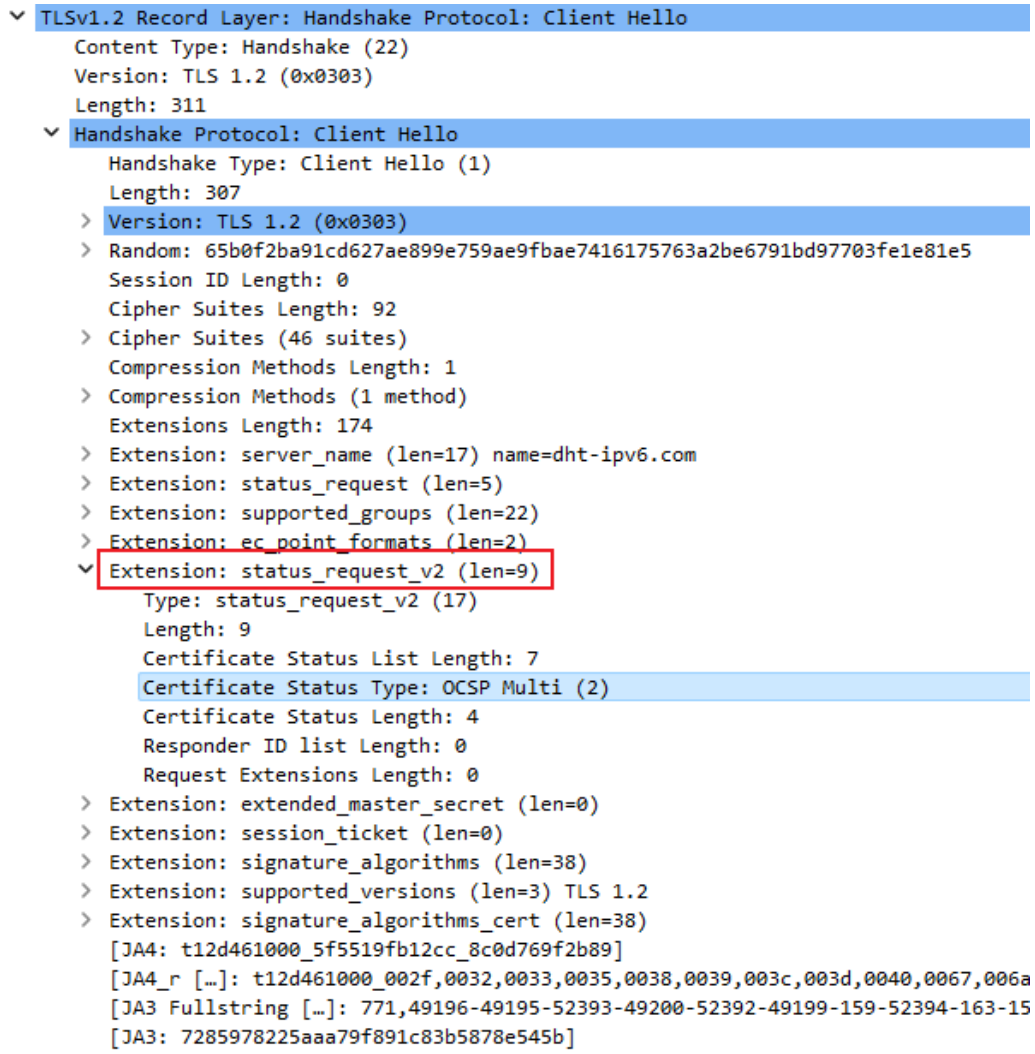
Figure 3-185 TLS-Certificate Status unknown



NOTE

The server returns certificate status only when the **Client Hello** packet sent by the client carries the **status_request** extended field.

Figure 3-186 status request



Step 5 In the navigation pane, choose **Devices** > **All Devices**. On the displayed page, find the target device to access its details page. Click the **Message Trace** tab and enable message tracing. Use the MQTT simulator certificate for two-way authentication. Check the message tracing error details. If the device certificate has been revoked, use a new valid certificate for access. Ensure to promptly revoke any leaked certificates.

Figure 3-187 Device certificate - Revoked

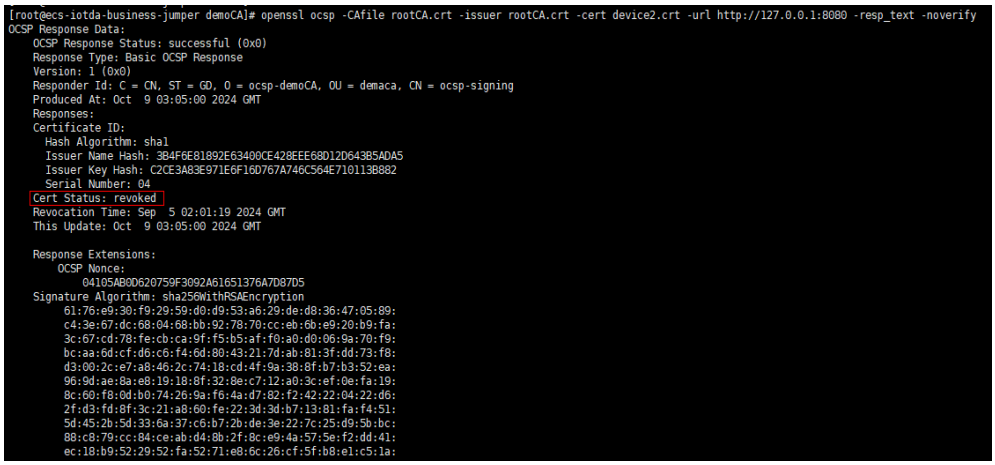
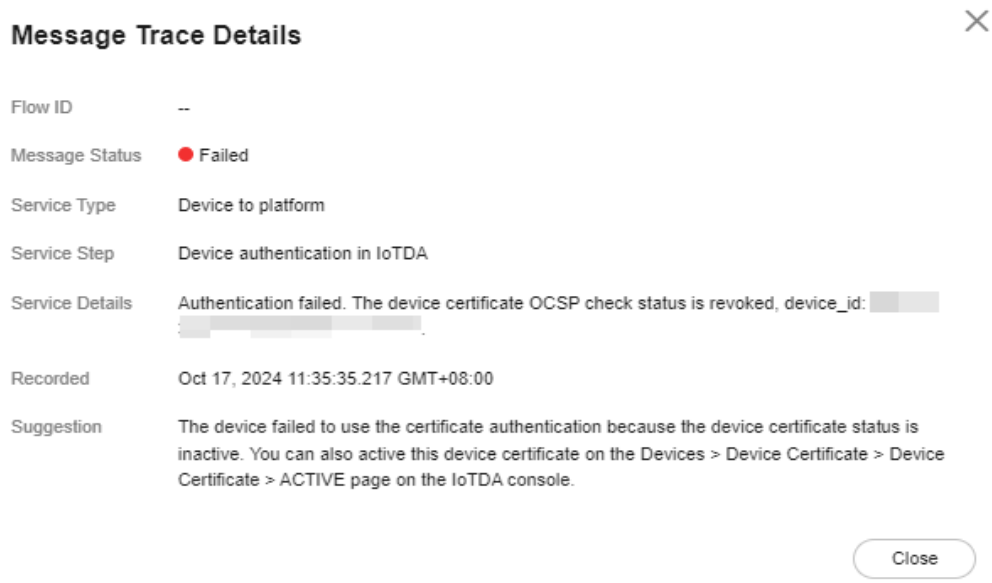


Figure 3-188 Message tracing - OCSP verification failure details



----End

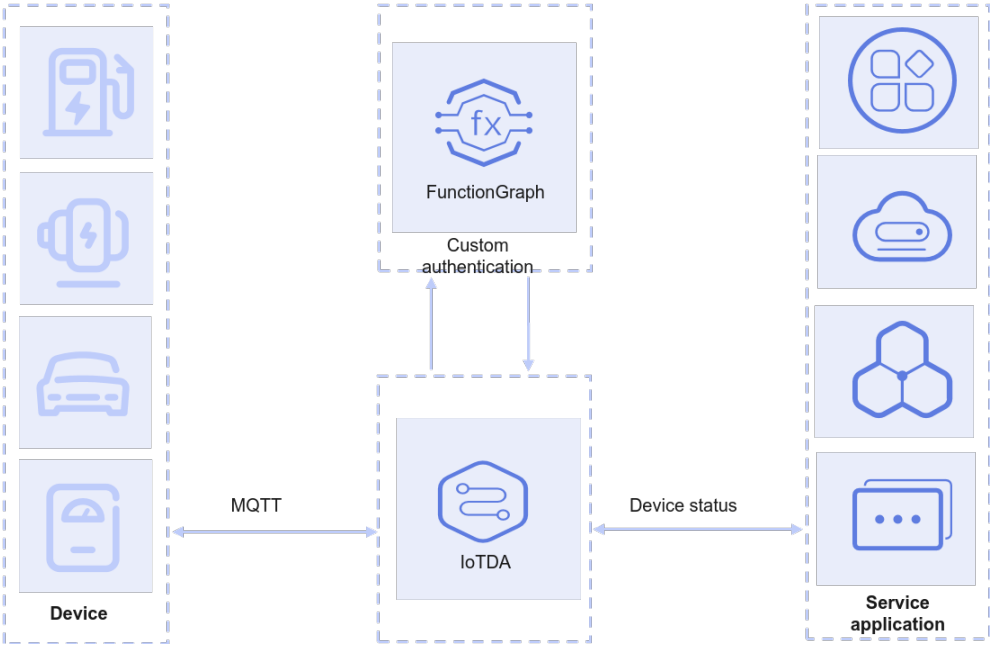
3.5.4 Custom Authentication

Introduction

You can use FunctionGraph to implement custom authentication logic to authenticate devices connected to the platform.

Before connecting a device to the platform, you can use the application to configure custom authentication on the console, and then configure related functions by using **FunctionGraph**. When the device connects to the platform, the platform obtains parameters such as the device ID and custom authentication function name, and sends an authentication request to FunctionGraph. You implement the authentication logic to complete access authentication.

Figure 3-189 Custom authentication architecture



Scenarios

- Device migration from third-party cloud platforms to IoTDA: You can configure the custom logic to make it compatible with the original authentication mode. No modification is required on the device side.
- Native access: Custom authentication logics are available for multiple scenarios.

Constraints

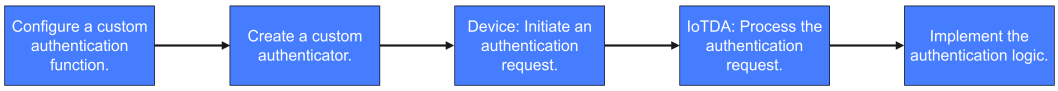
- The device must use TLS and support **Server Name Indication (SNI)**. The SNI must carry the domain name allocated by the platform.
- By default, each user can configure up to 10 custom authenticators.
- Max. processing time: 5 seconds. If the function does not return any result within 5 seconds, the authentication fails.
- For the TPS limit of each user, see **Product Specifications**. The TPS limit of custom authentication is 50% of the total authentication TPS (excluding device self-registration).
- If you have enabled the function of caching FunctionGraph authentication results, the modification takes effect only after the cache expires.
- The custom authentication mode is preferentially used for device access if conditions are met, for example, the custom authenticator name carried by the device is matched or a default custom authenticator has been configured.

NOTE

Our custom authentication mode enables device access without requiring reconstruction on the device side. It is important to avoid weak or verification-free modes. If the security level of your custom template is too low, it may lead to security issues. The platform does not assume any security responsibilities in such cases.

Process

Figure 3-190 Custom authentication process



Procedure

Step 1 Use **FunctionGraph** to create a custom authentication function. Access the console, search for **FunctionGraph**, and create a function.

Figure 3-191 Function list - Creating a function

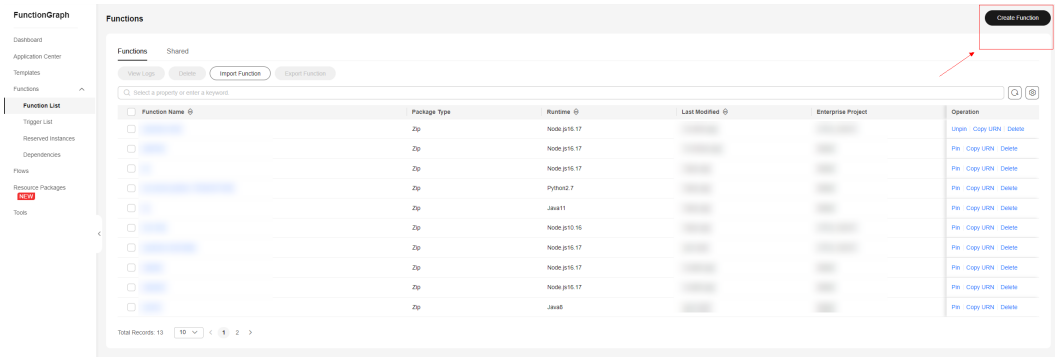
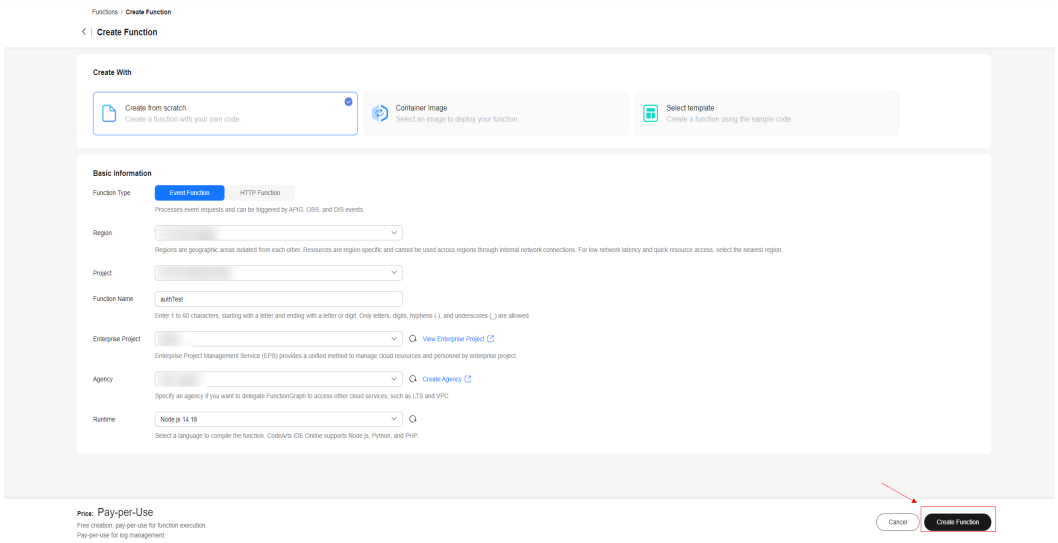


Figure 3-192 Creating a function - Parameters



Step 2 Configure custom authentication on the console for storage, management, and maintenance. Max. 10 custom authenticators can be configured.

Figure 3-193 Custom authentication - Entry

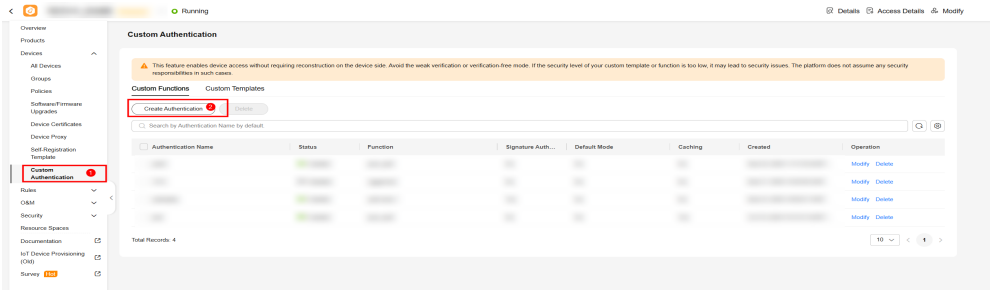


Figure 3-194 Custom authentication - Creating an authenticator

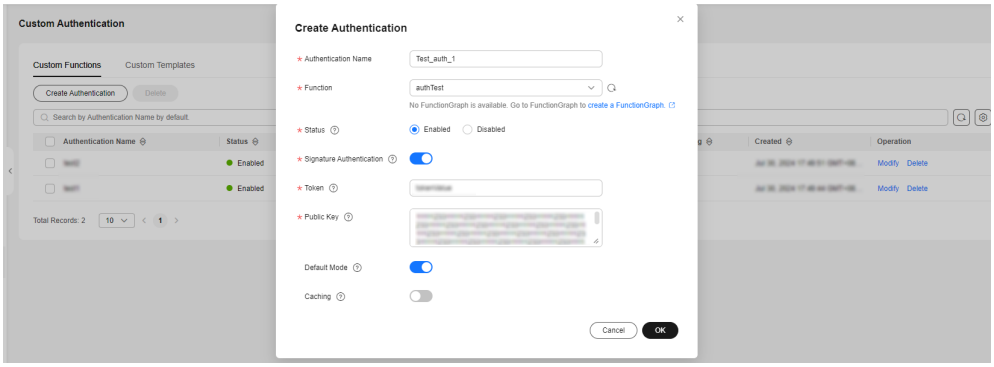


Table 3-28 Custom authentication parameters

Parameter	Mandatory	Description
Authentication Name	Yes	Enter a custom authenticator name.
Function	Yes	Select the corresponding function from the list created with FunctionGraph in **Step 1**.
Status	Yes	To use an authenticator, you must first enable it as it is disabled by default.
Signature Authentication	Yes	After this function is enabled (by default), authentication information that does not meet signature requirements will be rejected to reduce invalid function calls.
Token	No	Token for signature authentication. Used to check whether a device's signature information is valid.
Public Key	No	Public key for signature authentication. Used to check whether a device's signature information is valid.
Default Mode	Yes	After this function is enabled (disabled by default), if the username in an authentication request does not contain the **authorizer_name** parameter, this authenticator is used.
Caching	Yes	Whether to cache FunctionGraph authentication results (disabled by default). The cache duration ranges from 300 minutes to 1 day.

Step 3 The device initiates a CONNECT request using MQTT. The request must carry the **username** parameter, which contains optional parameters related to custom authentication.

- Username format requirements: Remove braces ({}) and separate each parameter by a vertical bar (|). Do not add vertical bars (|) in the parameter content.
  {**device-identifier**}|authorizer-name={**authorizer-name**}|authorizer-signature={**token-signature**}|signing-token={**token-value**}
  Example:
  659b70a0bd3f665a471e5ec9_auth|authorizer-name=Test_auth_1|authorizer-signature=***|signing-token=tokenValue

  Table 3-29 Description of the username parameter

Parameter	Mandatory	Description
device-identifier	Yes	Device identifier. You are advised to set it to the device ID.
authorizer-name	No	Custom authenticator name, which must be the same as the configured authenticator. If this parameter is not carried, the system will use either the default custom authenticator (if configured) or the original secret/certificate authentication mode.
authorizer-signature	No	This parameter is mandatory when the signature verification function is enabled. Obtain the value by encrypting the private key and signing-token. The value must be the same as the authentication name used in **Step 2**.
signing-token	No	This parameter is mandatory when the signature verification function is enabled. The value is used for signature verification and must be the same as the token value used in **Step 2**.

- Run the following command to obtain **authorizer-signature**:
  echo -n {signing-token} | openssl dgst -sha256 -sign {private key} | openssl base64

  Table 3-30 Command parameters

Parameter	Description
echo -n {signing-token}	Run the echo command to output the value of signing-token and use the **-n** parameter to remove the newline character at the end. The value of signing-token must be the same as that of the token in **Step 2**.
openssl dgst -sha256 -sign	Hash the input data with the SHA-256 algorithm.

Parameter	Description
{private key}	Private key encrypted using the RSA algorithm. You can upload a private key file in .pem or .key format.
openssl base64	Encode the signature result using Base64 for transmission and storage.

Step 4 When receiving an authentication request, IoTDA determines whether to use the custom authentication mode based on the username parameter and related configuration.

1. The system checks whether the username carries the custom authentication name. If yes, the authenticator processing function is matched based on the name. If no, the default custom authenticator is used to match the authentication processing function. If no matching is found, the original key/certificate authentication mode is used.
2. The system checks whether signature verification is enabled. If yes, the system checks whether the signature information carried in the username can be verified. If the verification fails, an authentication failure message is returned.
3. After the function matching, the system sends an authentication request to FunctionGraph using the Uniform Resource Name (URN) of the function and the device authentication information (the input parameter **event** in **Step 5**).

Step 5 Develop based on the processing function created with FunctionGraph in **Step 1**. Example for using the function and the JSON format of the returned result:

```
exports.handler = async (event, context) => {
    console.log("username=" + event.username);
    // Enter the validation logic.


    // Returned JSON format (fixed)
    const authRes = {
        "result_code": 200,
        "result_desc": "successful",
        "refresh_seconds": 300,
        "device": {
            "device_id": "myDeviceId",
            "provision_enable": true,
            "provisioning_resource": {
                "device_name": "myDeviceName",
                "node_id": "myNodeId",
                "product_id": "myProductId",
                "app_id": "customization0000000000000000000",
                "policy_ids": ["657a4e0c2ea0cb2cd831d12a", "657a4e0c2ea0cb2cd831d12b"]
            }
        }
    }
    return JSON.stringify(authRes);
}
```

Request parameters (**event**, in JSON format) of the function:

```
{
    "username": "myUserName",
    "password": "myPassword",
    "client_id": "myClientId",
    "certificate_info": {
        "common_name": "",
```



```
    "fingerprint": "123"
  }
}
```

Table 3-31 Request parameters

Parameter	Type	Mandatory	Description
username	String	Yes	The **username** field in the MQTT CONNECT message. Its format is the same as that of the **username** field in **Step 3**.
password	String	Yes	**password** parameter in the MQTT CONNECT message.
client_id	String	Yes	**clientId** parameter in the MQTT CONNECT message.
certificate_info	JsonObject	No	Device certificate information in the MQTT CONNECT message.

Table 3-32 certificate_info parameters

Parameter	Type	Mandatory	Description
common_name	String	Yes	Common name parsed from the device certificate carried by the device.
fingerprint	String	Yes	Fingerprint information parsed from the device certificate carried by the device.

Table 3-33 Returned parameters

Parameter	Type	Mandatory	Description
result_code	Integer	Yes	Authentication result code. If 200 is returned, the authentication is successful.
result_desc	String	No	Description of the authentication result.
refresh_seconds	Integer	No	Cache duration of the authentication result, in seconds.

Parameter	Type	Mandatory	Description
device	JsonObject	No	Device information when the authentication is successful. If the device ID in the device information does not exist and **device self-registration** is enabled, the platform automatically creates a device based on the device information.

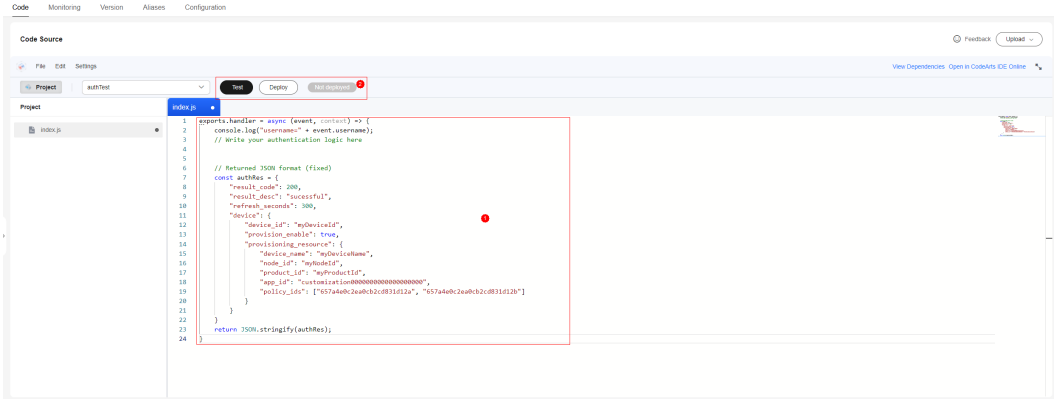
Table 3-34 Device parameters

Parameter	Type	Mandatory	Description
device_id	String	Yes	**Definition:** Globally unique device ID. Mandatory in both self-registration and non-self-registration scenarios. If this parameter is carried, the platform sets the device ID to the value of this parameter. Recommended format: *product_id_node_id*. **Range:** The value can contain up to 128 characters. Only letters, digits, underscores (_), and hyphens (-) are allowed. You are advised to use at least 4 characters.
provision_enable	Boolean	No	**Definition:** Whether to enable self-registration. Default value: **false**.
provisioning_resource	JsonObject	Mandatory in the self-registration scenario	**Definition:** Self-registration parameters.

Table 3-35 provisioning_resource self-registration parameters

Parameter	Type	Mandatory	Description
device_name	String	No	**Definition:** Device name, which uniquely identifies a device in a resource space. **Range:** The value can contain up to 256 characters. Only letters, digits, and special characters (_?'#().,&%@!-) are allowed. You are advised to use at least 4 characters. Min. characters: 1 Max. characters: 256
node_id	String	Yes	**Definition:** Device identifier. This parameter is set to the IMEI, MAC address, or serial number. It contains 1 to 64 characters, including letters, digits, hyphens (-), and underscores (_). (Note: Information cannot be modified once it is hardcoded to NB-IoT modules. Therefore, the node ID of an NB-IoT must be globally unique.) **Range:** The value can contain up to 64 characters. Only letters, digits, underscores (_), and hyphens (-) are allowed. You are advised to use at least 4 characters.
product_id	String	Yes	**Definition:** Unique ID of the product associated with the device. The value is allocated by IoTDA after the product is created. **Range:** The value can contain up to 256 characters. Only letters, digits, and special characters (_?'#().,&%@!-) are allowed. You are advised to use at least 4 characters. Min. characters: 1 Max. characters: 256
app_id	String	Yes	**Definition:** Resource space ID, which specifies the resource space to which the created device belongs. **Range:** The value can contain up to 36 characters. Only letters, digits, underscores (_), and hyphens (-) are allowed.
policy_ids	List<String>	No	**Definition:** Topic policy ID.

Figure 3-195 Compiling a function - Deployment



Step 6 After receiving the result, FunctionGraph checks whether the self-registration is required. If yes, FunctionGraph triggers automatic device registration. By default, all self-registered devices are authenticated using secrets, which are randomly generated. After receiving the authentication result, IoTDA proceeds with the subsequent process.

----End

3.5.5 Custom-Template Authentication

3.5.5.1 Usage

Introduction

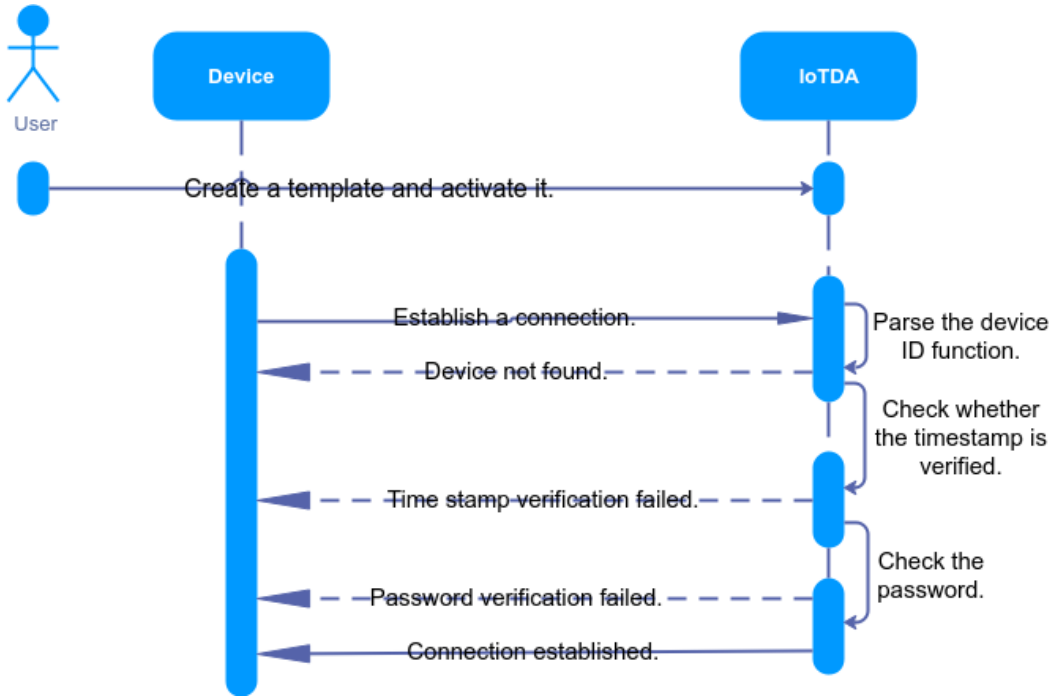
In addition to **the default authentication mode,** you can also use the **internal functions** provided by the platform to flexibly orchestrate authentication modes for devices connecting to the platform.

Scenarios

- Device migration from third-party IoT platforms to IoTDA: You can configure a custom template to be compatible with the original authentication mode. No modification is required on the device side.
- Native access: Custom templates can support more devices.

Process

Figure 3-196 Process of authentication based on custom templates



Constraints

1. The device must use TLS and support **Server Name Indication (SNI)**. The SNI must carry the domain name allocated by the platform.
2. Max. templates: five for a user. Only one template can be enabled at a time.
3. Max. functions nested: five layers.
4. Max. content length: 4,000 characters. Chinese character not allowed.
5. When the device uses secret authentication, the template password function must contain the original secret parameter (**iotda::device:secret**).
6. The format of the template authentication parameter **username** cannot be the same as that of the custom function authentication parameter **username**. Otherwise, the custom function authentication is used. For example:
   ```
   {deviceId}|authorizer-name={authorizer-name}|xxx
   ```
7. As custom authentication templates have higher priority, once you activate a custom authentication template, the platform uses the template instead of the default mode.

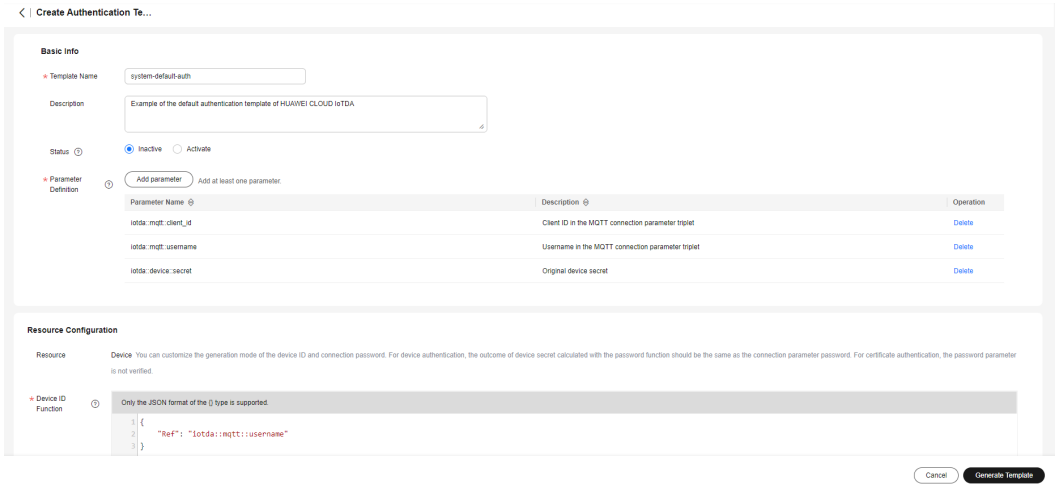
NOTE

Our custom authentication mode enables device access without requiring reconstruction on the device side. It is important to avoid weak or verification-free modes. If the security level of your custom template is too low, it may lead to security issues. The platform does not assume any security responsibilities in such cases.

Procedure

Step 1 Create an authentication template. Specifically, log in to the IoTDA console, in the navigation pane, choose **Devices** > **Custom Authentication**, click **Custom Template**, and click **Create Template**. The authentication template used in this example is the same as that used in the **default authentication**.

Figure 3-197 Custom authentication - Creating a template



The overall content of the template is as follows:

```
{
  "template_name": "system-default-auth",
  "description": "Example of the default authentication template of Huawei Cloud IoTDA",
  "status": "ACTIVE",
  "template_body": {
    "parameters": {
      "iotda::mqtt::client_id": {
        "type": "String"
      },
      "iotda::mqtt::username": {
        "type": "String"
      },
      "iotda::device::secret": {
        "type": "String"
      }
    },
    "resources": {
      "device_id": {
        "Ref": "iotda::mqtt::username"
      },
      "timestamp": {
        "type": "FORMAT",
        "pattern": "yyyyMMddHH",
        "value": {
          "Fn::SubStringAfter": [
            "${iotda::mqtt::client_id}",
            "_0_1_"
          ]
        }
      },
      "password": {
        "Fn::HmacSHA256": [
          "${iotda::device::secret}",
          {
            "Fn::SubStringAfter": [
              "${iotda::mqtt::client_id}",
```

```
          "_0_1_"
        ]
      }
    ]
  }
 }
}
```

Table 3-36 Authentication template parameters

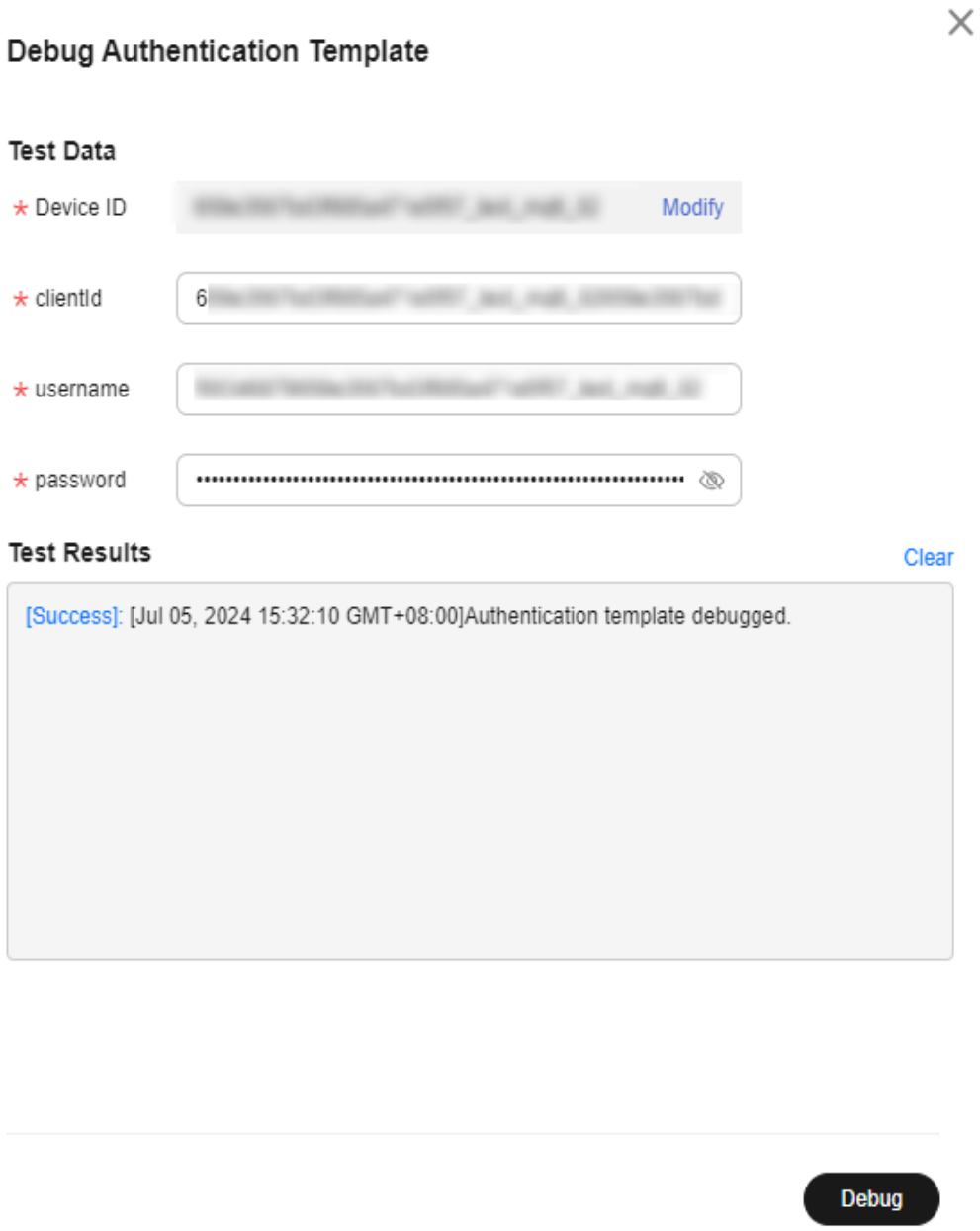
Parameter	Item	Mandatory	Description
template_name	Template name	Yes	Template name. The name must be unique for a single user. Max. length: 128 characters. Use only letters, digits, underscores (_), and hyphens (-).
description	Description	No	Template description. Max. length: 2,048 characters. Use only letters, digits, and special characters (_?'#().,&%@!-).
status	Status	No	Template status. By default, a template is not enabled. A user can only have one enabled template at a time.

Parameter	Item	Mandatory	Description
parameters	Parameter	Yes	MQTT connection parameters predefined by the platform. When a device uses password authentication, the template must contain the original secret parameter (**iotda::device:secret**). The platform predefines the following parameters: **iotda::mqtt::client_id**: **Client Id** in the MQTT connection parameter triplet **iotda::mqtt::username**: **User Name** in the MQTT connection parameter triplet **iotda::certificate::country**: device certificate (country/region, C) **iotda::certificate::organization**: device certificate (organization, O) **iotda::certificate::organizational_unit**: device certificate (organization unit, OU) **iotda::certificate::distinguished_name_qualifier**: device certificate (distinguishable name qualifier, dnQualifier) **iotda::certificate::state_name**: device_certificate (province/city, ST) **iotda::certificate::common_name**: device certificate (common name, CN) **iotda::certificate::serial_number**: device certificate (serial number, serialNumber) **iotda::device::secret**: original secret of the device
device_id	Device ID function	Yes	Function for obtaining the device ID, in JSON format. The platform parses this function to obtain the corresponding device information.
timestamp	Timestamp verification	No	Whether to verify the timestamp in the device connection information. Recommended: Enable this function if the device connection parameters (**clientId** and **username**) contain the timestamp. Verification process: The platform compares the timestamp carried by the device with the platform system time. If the timestamp plus 1 hour is less than the platform system time, the verification fails.

Parameter	Item	Mandatory	Description
type	Timestamp type	No	**UNIX**: Unix timestamp. Long integer, in seconds. **FORMAT**: formatted timestamp, for example, **2024-03-28 11:47:39** or **2024/03/28 03:49:13**.
pattern	Timestamp format	No	Time format template. Mandatory when the timestamp type is **FORMAT**. **y**: year **M**: month **d**: day **H**: hour **m**: minute **s**: second **S**: millisecond Example: **yyyy-MM-dd HH:mm:ss** and **yyyy/MM/dd HH:mm:ss**
value	Timestamp function	No	Function for obtaining the timestamp when the device establishes a connection. Mandatory when timestamp verification is enabled.
password	MQTT password function	No	Password function. Mandatory when the device authentication type is secret authentication. The template parameters must contain the original device secret parameter (**iotda::device:secret**). For details about the device authentication type, see **Registering an Individual Device**. Verification process: The platform uses parameters such as the original secret of the device in the function to calculate. If the result is the same as the **password** carried in the connection establishment request, the authentication is successful. Otherwise, the authentication fails.

Step 2 Select a device debugging template. Click **Debug**, select a device for debugging, enter MQTT connection parameters, and click **Debug** to check the result. Note: If **clientId** in the standard format is used, the platform verifies whether the value of **username** is the same as the prefix of **clientId**.

Figure 3-198 Custom template - Debugging



After the device debugging is successful, click **Enable** to enable the template. Once the template is enabled, it will be used for authentication of all devices, and the enabled template cannot be modified. You are advised to make modification on the copy of the target template and debug it. Switch to the modified template only after the debugging is successful.

Step 3 Use MQTT.fx to simulate device connection setup. Set **Broker Address** to **the platform access address**, choose **Overview** > **Access Information**, and set port to **8883**.

Figure 3-199 Device connection establishment

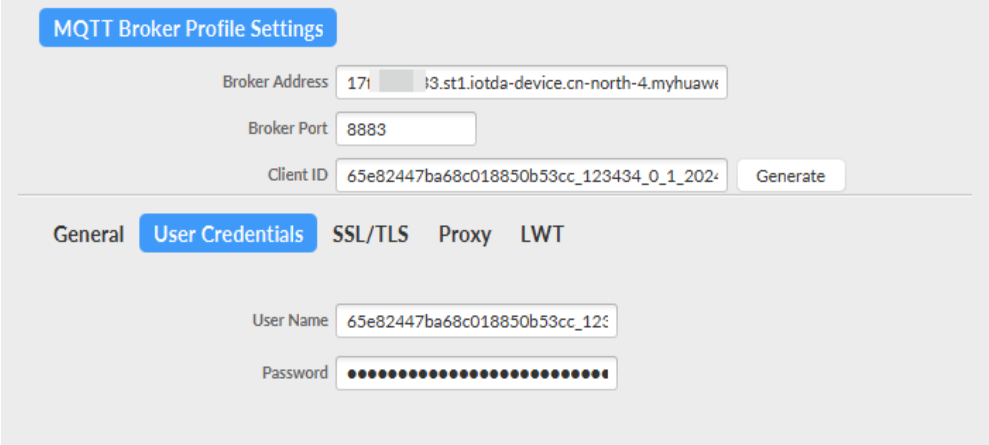
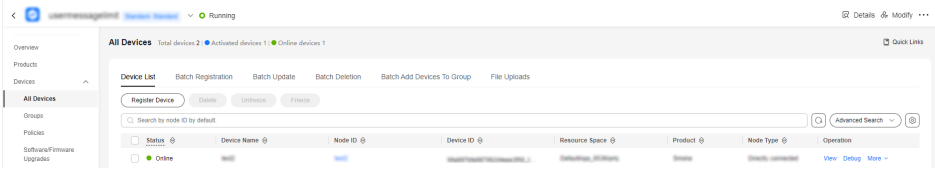


Figure 3-200 Device list - Device online status



----End

3.5.5.2 Examples

Example 1

When a certificate is used to authenticate a device, the values of **UserName** and **ClientId** are not limited. The device ID is obtained from the common name of the device certificate.

Table 3-37 Authentication parameters

Parameter	Description
Client ID	Any value
User Name	Any value
Password	Empty value

Authentication template:

```
{
  "template_name": "template1",
  "description": "template1",
  "template_body": {
    "parameters": {
      "iotda::certificate::common_name": {
        "type": "String"
      }
    },
```



```
    "resources": {
      "device_id": {
        "Ref": "iotda::certificate::common_name"
      }
    }
  }
}
```

Example 2

The device ID follows the format of *${ProductId}_${NodeId}* and the authentication parameters are as outlined in the table below.

Table 3-38 Authentication parameters

Parameter	Description
Client ID	Fixed format: **${ClientId}\|securemode=2,signmethod=hmacsha256,timestamp=${timestamp}|** • *${ClientId}* (fixed format): *${ProductId}.${NodeId}* – *${NodeId}*: device node ID – *${ProductId}*: product ID • *${timestamp}*: Unix timestamp, in milliseconds
User Name	Fixed format: **${NodeId}&${ProductId}**
Password	Result value after encrypting the combination of device parameter and parameter value, with the device password as the key and HMAC-SHA256 algorithm as the tool. Encryption string format: **clientId${clientId}deviceName${nodeId}productKey${productId}timestamp${timestamp}** • *${ClientId}* (fixed format): *${ProductId}.${NodeId}* • *${NodeId}*: device node ID • *${ProductId}*: product ID • *${timestamp}*: timestamp

Authentication template:

```
{
  "template_name": "template2",
  "description": "template2",
  "template_body": {
    "parameters": {
      "iotda::mqtt::client_id": {
        "type": "String"
      },
      "iotda::mqtt::username": {
        "type": "String"
      },
      "iotda::device::secret": {
        "type": "String"
      }
    },
    "resources": {
```



```
        "device_id": {
            "Fn::Join": [{
                "Fn::SplitSelect": [
                    "${iotda::mqtt::username}",
                    "&",
                    1
                ]
            }, "_", {
                "Fn::SplitSelect": [
                    "${iotda::mqtt::username}",
                    "&",
                    0
                ]
            }]
        },
        "timestamp": {
            "type": "UNIX",
            "value": {
                "Fn::MathDiv": [{
                    "Fn::ParseLong": {
                        "Fn::SplitSelect": [{
                            "Fn::SubStringAfter": [{
                                "Fn::SplitSelect": ["${iotda::mqtt::client_id}", "|", 1]
                            }, "timestamp="]
                        }, ",", 0]
                    }
                }, 1000]
            }
        },
        "password": {
            "Fn::HmacSHA256": [{
                    "Fn::Sub": [
                        "clientId${clientId}deviceName${deviceName}productKey${productKey}timestamp${timestamp}",
                        {
                            "clientId": {
                                "Fn::SplitSelect": [
                                    "${iotda::mqtt::client_id}",
                                    "|",
                                    0
                                ]
                            },
                            "deviceName": {
                                "Fn::SplitSelect": [
                                    "${iotda::mqtt::username}",
                                    "&",
                                    0
                                ]
                            },
                            "productKey": {
                                "Fn::SplitSelect": [
                                    "${iotda::mqtt::username}",
                                    "&",
                                    1
                                ]
                            },
                            "timestamp": {
                                "Fn::SplitSelect": [{
                                    "Fn::SubStringAfter": [{
                                        "Fn::SplitSelect": ["${iotda::mqtt::client_id}", "|", 1]
                                    }, "timestamp="]
                                }, ",", 0]
                            }
                        }
                    ]
                },
                "${iotda::device::secret}"
            ]
        }
```



```
    }
  }
}
```

Example 3

The device ID follows the format of *${productId}_${nodeId}* and the authentication parameters are as outlined in the table below.

Table 3-39 Authentication parameters

Parameter	Description
Client ID	Fixed format: **${productId}${nodeId}** ● *${productId}*: product ID ● *${nodeId}*: node ID
User Name	Fixed format: **${productId}${nodeId};12010126;${connid};${expiry}** ● *${productId}*: product ID ● *${nodeId}*: node ID ● *${connid}*: random string ● *${expiry}*: Unix timestamp, in seconds
Password	Fixed format: **${token};hmacsha256** ● *${token}*: result value after encrypting the **User Name** field, with the HMAC-SHA256 algorithm as the tool and the Base64-decoded device password as the key.

Authentication template:

```
{
  "template_name": "template3",
  "description": "template3",
  "template_body": {
    "parameters": {
      "iotda::mqtt::client_id": {
        "type": "String"
      },
      "iotda::mqtt::username": {
        "type": "String"
      },
      "iotda::device::secret": {
        "type": "String"
      }
    },
    "resources": {
      "device_id": {
        "Ref": "iotda::mqtt::client_id"
      },
      "timestamp": {
        "type": "UNIX",
        "value": {
          "Fn::ParseLong": {
            "Fn::SplitSelect": ["${iotda::mqtt::username}", ";", 3]
          }
```



```
            }
        },
        "password": {
            "Fn::Sub": [
                "${token};hmacsha256",
                {
                    "token": {
                        "Fn::HmacSHA256": [
                            "${iotda::mqtt::username}",
                            {
                                "Fn::Base64Decode": "${iotda::device::secret}"
                            }
                        ]
                    }
                }
            ]
        }
    }
  }
}
```

3.5.5.3 Internal Functions

Introduction

Huawei Cloud IoTDA provides multiple internal functions to use in templates. This section introduces these functions, including the input parameter type, parameter length, and return value type.

NOTE

- The entire function must be in valid JSON format.
- In a function, the variable placeholders (${}) or the **Ref** function can be used to reference the value defined by the input parameter.
- The parameters used by the function must be declared in the template.
- A function with a single input parameter is followed by a parameter, for example, **"Fn::Base64Decode": "***${iotda::mqtt::username}***"**.
- A function with multiple input parameters is followed by an array, for example, **"Fn::HmacSHA256": ["***${iotda::mqtt::username}***", "***${iotda::device::secret}***"]**.
- Functions can be nested. That is, the parameter of a function can be another function. Note that the return value of a nested function must match its parameter type in the outer function, for example, **{"Fn::HmacSHA256": ["***${iotda::mqtt::username}***", {"Fn::Base64Encode": "***${iotda::device::secret}***"}]}**.
- The hash function (**Fn::HmacSHA256**) can be used twice at most in an authentication template.
- The total number of Base64 functions (**Fn::Base64Decode** and **Fn::Base64Encode**) in an authentication template cannot exceed 2.
- After applying the HmacSHA256 function to the password in the authentication template, the functions **Fn::Split**, **Fn::SplitSelect**, **Fn::SubStringAfter**, and **Fn::SubStringBefore** cannot be executed.

Fn::ArraySelect

The internal function **Fn::ArraySelect** returns a string element whose index is **index** in a string array.

JSON

```
{"Fn::ArraySelect": [index, [StringArray]]}
```


Table 3-40 Parameters

Parameter	Type	Description
index	int	Index of an array element. The value is an integer and starts from 0.
StringArray	String[]	String array element.
Return value	String	Element whose index is **index**.

Example:

```
{
    "Fn::ArraySelect": [1, ["123", "456", "789"]]
}
return: "456"
```

Fn::Base64Decode

The internal function **Fn::Base64Decode** decodes a string into a byte array using Base64.

JSON

```
{ "Fn::Base64Decode" : "content" }
```

Table 3-41 Parameters

Parameter	Type	Description
content	String	String to be decoded.
Return value	byte[]	Base64-decoded byte array.

Example:

```
{
    "Fn::Base64Decode": "123456"]
}
return: d76df8e7 // The value is converted into a hexadecimal string for display.
```

Fn::Base64Encode

The internal function **Fn::Base64Encode** encodes a string using Base64.

JSON

```
{"Fn::Base64Encode": "content"}
```

Table 3-42 Parameters

Parameter	Type	Description
content	String	String to be encoded.

Parameter	Type	Description
Return value	String	Base64-encoded string.

Example:

```
{
   "Fn::Base64Encode": "testvalue"
}
return: "dGVzdHZhbHVl"
```

Fn::GetBytes

The internal function **Fn::GetBytes** returns a byte array encoded from a string using UTF-8.

JSON

```
{"Fn::GetBytes": "content"}
```

Table 3-43 Parameters

Parameter	Type	Description
content	String	String to be encoded.
Return value	byte[]	Byte array converted from a string encoded using UTF-8.

Example:

```
{
   "Fn::GetBytes": "testvalue"
}
return: "7465737476616c7565" // The value is converted into a hexadecimal string for display.
```

Fn::HmacSHA256

The internal function **Fn::HmacSHA256** encrypts a string using the HmacSHA256 algorithm based on a given secret.

JSON

```
{"Fn::HmacSHA256": ["content", "secret"]}
```

Table 3-44 Parameters

Parameter	Type	Description
content	String	String to be encrypted.
secret	String or byte[]	Secret key, which can be a string or byte array.

Parameter	Type	Description
Return value	String	Value encrypted using the HmacSHA256 algorithm.

Example:

```
{
    "Fn::HmacSHA256": ["testvalue", "123456"]
}
return: "0f9fb47bd47449b6ffac1be951a5c18a7eff694940b1a075b973ff9054a08be3"
```

Fn::Join

The internal function **Fn::Join** can concatenate up to 10 strings into one string.

JSON

```
{"Fn::Join": ["element", "element"...]}
```

Table 3-45 Parameters

Parameter	Type	Description
element	String	String to be concatenated.
Return value	String	String obtained by concatenating substrings.

Example:

```
{
    "Fn::Join": ["123", "456", "789"]
}
return: "123456789"
```

Fn::MathAdd

The internal function **Fn::MathAdd** performs mathematical addition on two integers.

JSON

```
{"Fn::MathAdd": [X, Y]}
```

Table 3-46 Parameters

Parameter	Type	Description
X	long	Augend.
Y	long	Augend.
Return value	long	Sum of X and Y.

Example:

```
{
   "Fn::MathAdd": [1, 1]
}
return: 2
```

Fn::MathDiv

The internal function **Fn::MathDiv** performs a mathematical division on two integers.

JSON

```
{"Fn::MathDiv": [X, Y]}
```

Table 3-47 Parameters

Parameter	Type	Description
X	long	Dividend.
Y	long	Divisor.
Return value	long	Value of X divided by Y.

Example:

```
{
   "Fn::MathDiv": [10, 2]
}
return: 5

{
   "Fn::MathDiv": [10, 3]
}
return: 3
```

Fn::MathMod

The internal function **Fn::MathMod** performs the mathematical modulo on two integers.

JSON

```
{"Fn::MathMod": [X, Y]}
```

Table 3-48 Parameters

Parameter	Type	Description
X	long	Dividend.
Y	long	Divisor.
Return value	long	Residue of X modulo Y.

Example:

```
{
   "Fn::MathMod": [10, 3]
}
return: 1
```

Fn::MathMultiply

The internal function **Fn::MathMultiply** performs mathematical multiplication on two integers.

JSON

```
{"Fn::MathMultiply": [X, Y]}
```

Table 3-49 Parameters

Parameter	Type	Description
X	long	Multiplicand.
Y	long	Multiplicand.
Return value	long	Value of X multiplied by Y.

Example:

```
{
   "Fn::MathMultiply": [3, 3]
}
return: 9
```

Fn::MathSub

The internal function **Fn::MathSub** performs mathematical subtraction on two integers.

JSON

```
{"Fn::MathSub": [X, Y]}
```

Table 3-50 Parameters

Parameter	Type	Description
X	long	Minuend.
Y	long	Subtrahend.
Return value	long	Value of X minus Y.

Example:

```
{
   "Fn::MathSub": [9, 3]
```



```
}
return: 6
```

Fn::ParseLong

The internal function **Fn::ParseLong** converts a numeric string into an integer.

JSON

```
{"Fn::ParseLong": "String"}
```

Table 3-51 Parameters

Parameter	Type	Description
String	String	String to be converted.
Return value	long	Value obtained after a string is converted into an integer.

Example:

```
{
   "Fn::ParseLong": "123"
}
return: 123
```

Fn::Split

The internal function **Fn::Split** splits a string into a string array based on the specified separator.

JSON

```
{ "Fn::Split" : ["String", "Separator"] }
```

Table 3-52 Parameters

Parameter	Type	Description
String	String	String to be split.
Separator	String	Separator.
Return value	String[]	String array obtained after **String** is split by **Separator**.

Example:

```
{
   "Fn::Split": ["a|b|c", "|"]
}
return: ["a", "b", "c"]
```


Fn::SplitSelect

The internal function **Fn::SplitSelect** splits a string into a string array based on the specified separator, and then returns the elements of the specified index in the array.

JSON

```
{ "Fn::SplitSelect" : ["String", "Separator", index] }
```

Table 3-53 Parameters

Parameter	Type	Description
String	String	String to be split.
Separator	String	Separator.
index	int	Index value of the target element in the array, starting from 0.
Return value	String	Substring of the specified index after a string is split by the specified separator.

Example:

```
{
   "Fn::SplitSelect": ["a|b|c", "|", 1]
}
return: "b"
```

Fn::Sub

The internal function **Fn::Sub** replaces variables in an input string with specified values. You can use this function in a template to construct a dynamic string.

JSON

```
{ "Fn::Sub" : [ "String", { "Var1Name": Var1Value, "Var2Name": Var2Value } ] }
```

Table 3-54 Parameters

Parameter	Type	Description
String	String	A string that contains variables. Variables are defined using placeholders (${}).
VarName	String	Variable name, which must be defined in the **String** parameter.
VarValue	String	Variable value. Function nesting is supported.
Return value	String	Value of string after replacement in the original **String** parameter.

Example:

```
{
   "Fn::Sub": ["${token};hmacsha256", {
      "token": {
         "Fn::HmacSHA256": ["${iotda::mqtt::username}", {
            "Fn::Base64Decode": "${iotda::mqtt::client_id}"
         }]
      }
   }]
}
If:
${iotda::mqtt::username}="test_device_username"
${iotda::device::client_id}="OozqTPlCWTTJjEH/5s+T6w=="
return: "0773c4fd6c92902a1b2f4a45fdcdec416b6fc2bc6585200b496e460e2ef31c3d"
```

Fn::SubStringAfter

The internal function **Fn::SubStringAfter** extracts a substring after a specified separator.

JSON

```
{ "Fn::SubStringAfter" : ["content", "separator"] }
```

Table 3-55 Parameters

Parameter	Type	Description
content	String	String to be extracted.
separator	String	Separator.
Return value	String	Substring after the specified separator that separates the string.

Example:

```
{
   "Fn::SubStringAfter": ["content:123456", ":"]
]
return: "123456"
```

Fn::SubStringBefore

The internal function **Fn::SubStringBefore** extracts a substring before a specified separator.

JSON

```
{ "Fn::SubStringBefore" : ["content", "separator"] }
```

Table 3-56 Parameters

Parameter	Type	Description
content	String	String to be extracted.

Parameter	Type	Description
separator	String	Separator.
Return value	String	Substring before the specified separator that separates the string.

Example:

```
{
   "Fn::SubStringBefore": ["content:123456", ":"]
]
return: "content"
```

Fn::ToLowerCase

The internal function **Fn::ToLowerCase** converts a string to the lowercase format.

JSON

```
{ "Fn::ToLowerCase" : content }
```

Table 3-57 Parameters

Parameter	Type	Description
content	String	String to be converted.
Return value	String	Value of a string after it is converted to the lowercase format.

Example:

```
{
   "Fn::ToLowerCase": "ABC"
]
return: "abc"
```

Fn::ToUpperCase

The internal function **Fn::ToUpperCase** converts a string to the uppercase format.

JSON

```
{ "Fn::ToUpperCase" : content }
```

Table 3-58 Parameters

Parameter	Type	Description
content	String	String to be converted.
Return value	String	Value of a string after it is converted to the uppercase format.

Example:

```
{
    "Fn::ToUpperCase": "abc"
]
return: "ABC"
```

Ref

The internal function **Ref** returns the value of the specified referenced parameter. The referenced parameter must be declared in the template.

JSON

```
{ "Ref" : "paramName" }
```

Table 3-59 Parameters

Parameter	Type	Description
paramName	String	Name of the referenced parameter.
Return value	String	Value of the referenced parameter.

Example:

```
{
    "Ref": "iotda::mqtt::username"
}
If iotda::mqtt::username="device_123"
return: "device_123"
```

3.6 HTTP(S) Access

Introduction

IoTDA supports HTTPS, a secure communication protocol derived from HTTP and secured with SSL encryption. HTTPS is commonly employed for data collection and analysis due to HTTP's efficiency in transmitting and processing structured data. Additionally, it is utilized in scenarios where devices require non-persistent connections and unidirectional data upload.

In HTTPS-based authentication, a device utilizes the HTTPS-based device authentication API to securely transmit the device ID and secret. The secret is encrypted using an algorithm. After the authentication is successful, the connection between the device and the platform is established, and the platform returns an access token.

Constraints

- An access token is required when HTTPS APIs for property reporting and message reporting are called.
- If an access token expires, you need to authenticate the device again to obtain an access token.

- If you obtain a new access token before the old one expires, the old access token will be valid for 30 seconds before expiration.

Table 3-60 Constraints

Description	Constraint
Supported HTTP version	HTTP 1.0 HTTP 1.1
Supported HTTPS	The platform supports only the HTTPS protocol. For details about how to download a certificate, see **Certificates**.
Supported TLS version	TLS 1.2
Body length	1 MB
API specifications	**Specifications**
Number of child devices of which properties can be reported by a gateway at a time	50
Data delivery	Not supported

Endpoints

For details about the platform endpoint, see **Platform Connection Information**.

NOTE

Use the endpoint of IoTDA and the HTTPS port number 443.

Process

Figure 3-201 HTTPS access authentication process

1. An application calls the API for registering a device. Alternatively, a user uses the IoTDA console to register a device.
2. The platform allocates a globally unique device ID and secret to the device.

   NOTE

   The secret can be defined during device registration. If no secret is defined, the platform allocates one.
3. When a device attempts to connect to the platform, the device calls the HTTPS device authentication API to send an access authentication request to the platform. The request carries the device ID and the secret generated using the HMACSHA256 algorithm. The secret is the value obtained after the

password allocated by the platform is signed using the timestamp as the key. For details, see **Huawei Cloud IoTDA MQTT ClientId Generator**.

4. If the authentication is successful, the platform returns a success message, and the device is connected to the platform.

Procedure

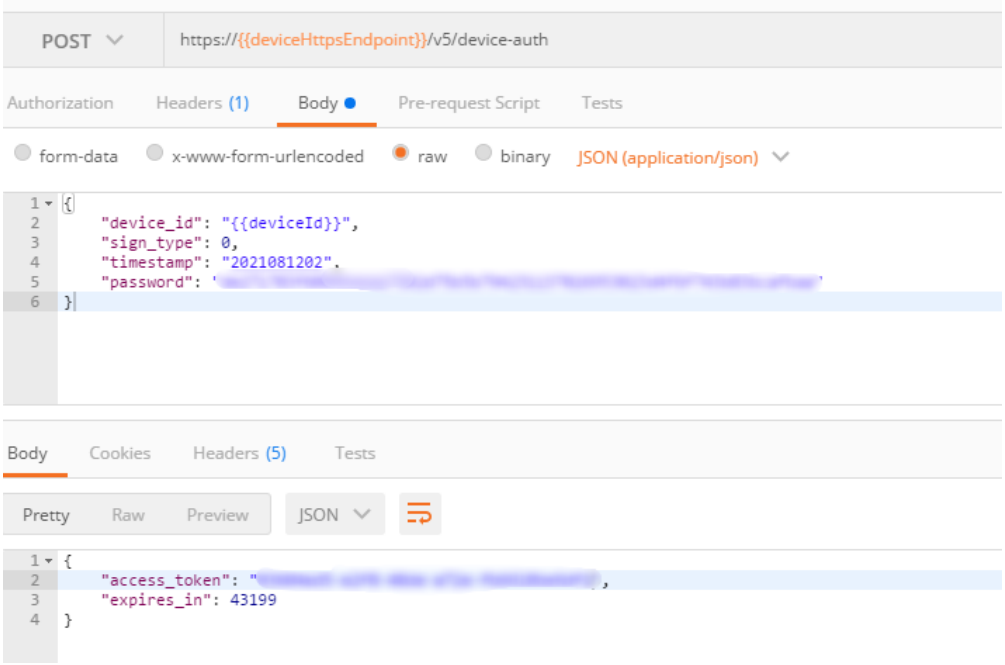
When a device connects to the platform through HTTPS, HTTPS APIs are used for their communication. These APIs can be used for device authentication as well as message and property reporting.

Table 3-61 Message type

Message Type	Description
Device authentication	Devices obtain access tokens.
Device property reporting	Devices report property data in the format defined in the product model.
Device message reporting	Devices report custom data to IoTDA, which then forwards reported messages to an application or other Huawei Cloud services for storage and processing.
Gateway batch property reporting	A gateway reports property data of multiple child devices to the platform.

1. Create a product on the IoTDA console or by calling the API for **creating a product**.
2. Register a device on the **IoTDA console** or calling the API for **creating a device**.
3. After the device is registered, obtain the access token of the device through the API for **device authentication**.

Figure 3-202 Obtaining the access token



4. Use the access token in the message header to report device **messages** or **properties**. The following figures use property reporting as an example.

Figure 3-203 Reporting properties

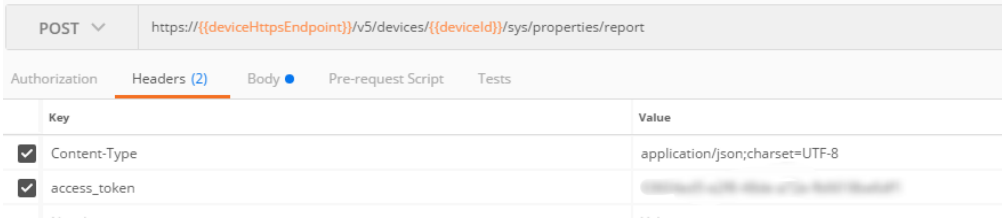
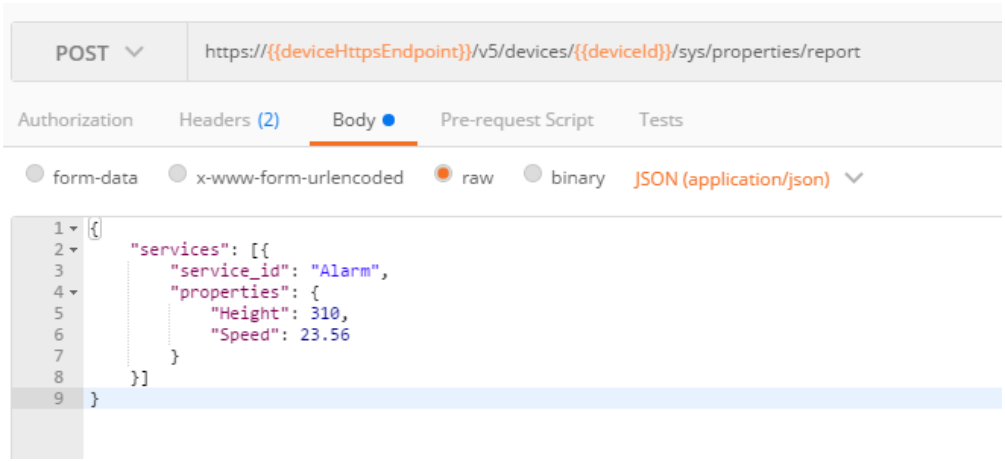


Figure 3-204 Reporting properties



3.7 LwM2M/CoAP Access

Introduction

Lightweight Machine to Machine (LwM2M), proposed by the Open Mobile Alliance (OMA), is a lightweight, standard, and universal IoT device management protocol that can be used to quickly deploy IoT services in client/server mode. LwM2M establishes a set of standards for IoT device management and application. It provides lightweight, compact, and secure communication interfaces and efficient data models for M2M device management and service support.

LwM2M/CoAP authentication supports both encrypted and non-encrypted access modes. Non-encrypted mode: Devices connect to IoTDA carrying the node ID through port 5683. Encrypted mode: Devices connect to IoTDA carrying node ID and secret through port 5684 by the DTLS/DTLS+ channel.

You are advised to use the encrypted access mode for security purposes.

NOTE

For details about LwM2M syntax and APIs, see **specifications**.

IoTDA supports the plain text, opaque, Core Link, TLV, and JSON encoding formats specified in the protocol. In the multi-field operation (for example, writing multiple resources), the TLV format is used by default.

Constraints

Table 3-62 Constraints on LwM2M/CoAP access

Description	Constraint
Supported LwM2M version	1.1
Supported DTLS version	DTLS 1.2
Supported cryptographic algorithm suite	TLS_PSK_WITH_AES_128_CCM_8 and TLS_PSK_WITH_AES_128_CBC_SHA256
Body length	1 KB
API specifications	**Specifications**

Endpoints

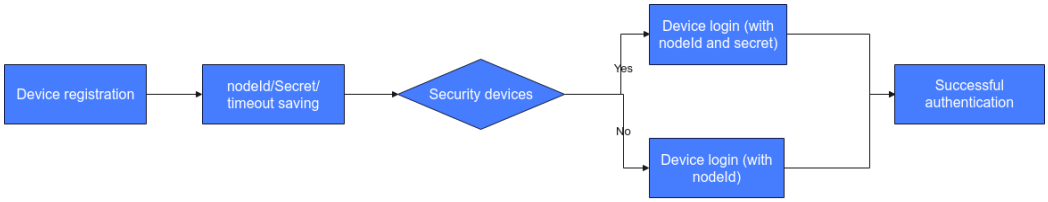
For details about the platform endpoint, see **Platform Connection Information**.

NOTE

Use the endpoint corresponding to CoAP (5683) or CoAPS (5684) and port 5683 (non-encrypted) or 5684 (encrypted) for device access.

Authentication Process

Figure 3-205 LwM2M/CoAP access authentication process



1. An application calls the API for registering a device. Alternatively, a user uses the IoTDA console to register a device.
2. The platform allocates a secret to the device and returns **timeout**.

   NOTE

   - The secret can be defined during device registration. If no secret is defined, the platform allocates one.
   - If the device is not connected to the platform within the duration specified by **timeout**, the platform deletes the device registration information.
3. During login, the device sends a connection authentication request carrying the node ID (such as the IMEI) and secret if it is a security device, or carrying the node ID if it is a non-security device.
4. If the authentication is successful, the platform returns a success message, and the device is connected to the platform.

Development Process

1. Development on the platform: Create products, develop product models and codecs on the platform, and register devices. For details, see **Creating a Product**, **Developing a Product Model**, **Developing a Codec**, and **Registering a Device**.
2. Development on the device: Use modules and Tiny SDKs on the device side for access. For details, see **IoT Device SDK Tiny (C) User Guide**.

Best Practices

Developing a Smart Street Light Using NB-IoT BearPi

FAQ

LwM2M/CoAP access FAQ:

- **How Do I Know the Strength of the NB-IoT Network Signal?**
- **What Do I Do If an NB-IoT Module Failed to Be Bound to a Device?**
- **What Do I Do If an NB-IoT Module Failed to Be Bound to a Device?**
- **What Can I Do If an NB-IoT Module Cannot Report Data?**
- **Why Was a 513 Message Reported During the Connection of an NB-IoT Device?**
- **Why Does Data Reporting Fails When an NB-IoT Card Is Used in Another Device?**

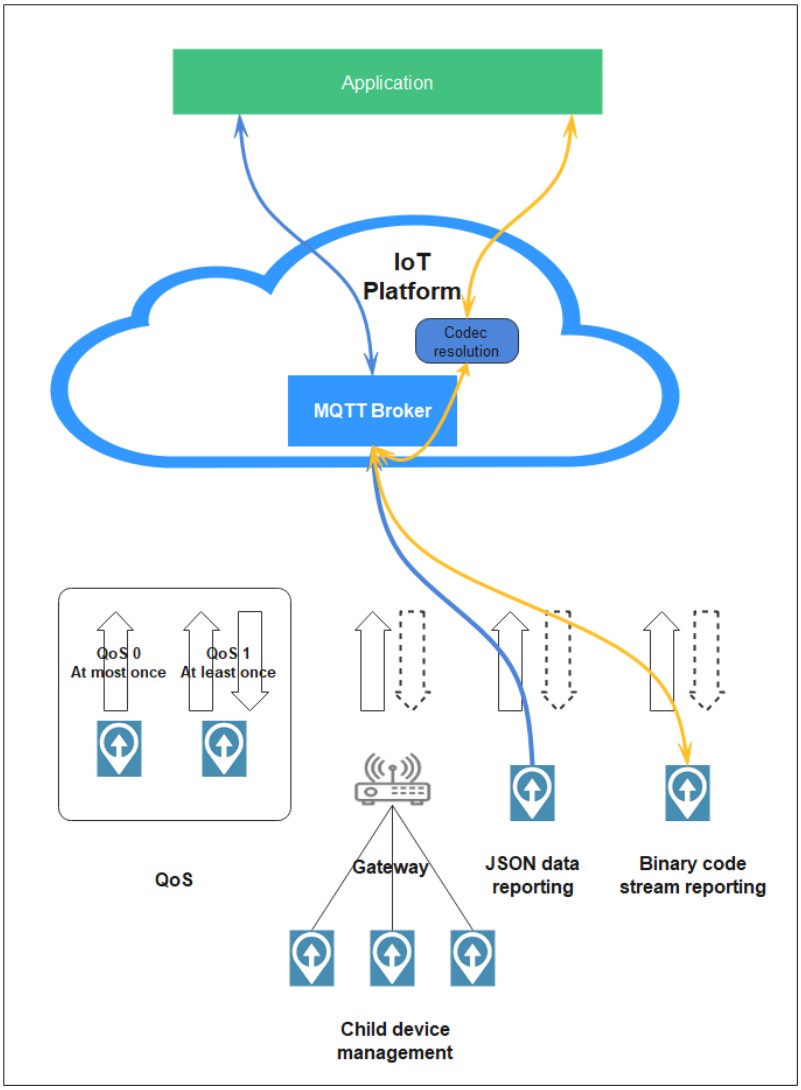
3.8 Access Using MQTT Demos

3.8.1 MQTT Usage Guide

Overview

Message Queuing Telemetry Transport (MQTT) is a publish/subscribe messaging protocol that transports messages between clients and servers. It is suitable for remote sensors and control devices (such as smart street lamps) that have limited computing capabilities and work in low-bandwidth, unreliable networks through persistent device-cloud connections. MQTT clients publish or subscribe to messages through topics. MQTT brokers centrally manage message routing and ensure end-to-end message transmission reliability based on the preset quality of service (QoS). In this process, the client that sends messages (publisher) and the client that receives messages (subscriber) are decoupled, eliminating the need for a direct connection between them. MQTT has emerged as a top protocol in the IoT domain by meeting the lightweight, reliable, bidirectional, and scalable communication protocol needs of IoT applications. To learn more about the MQTT syntax and interfaces, click **here**.

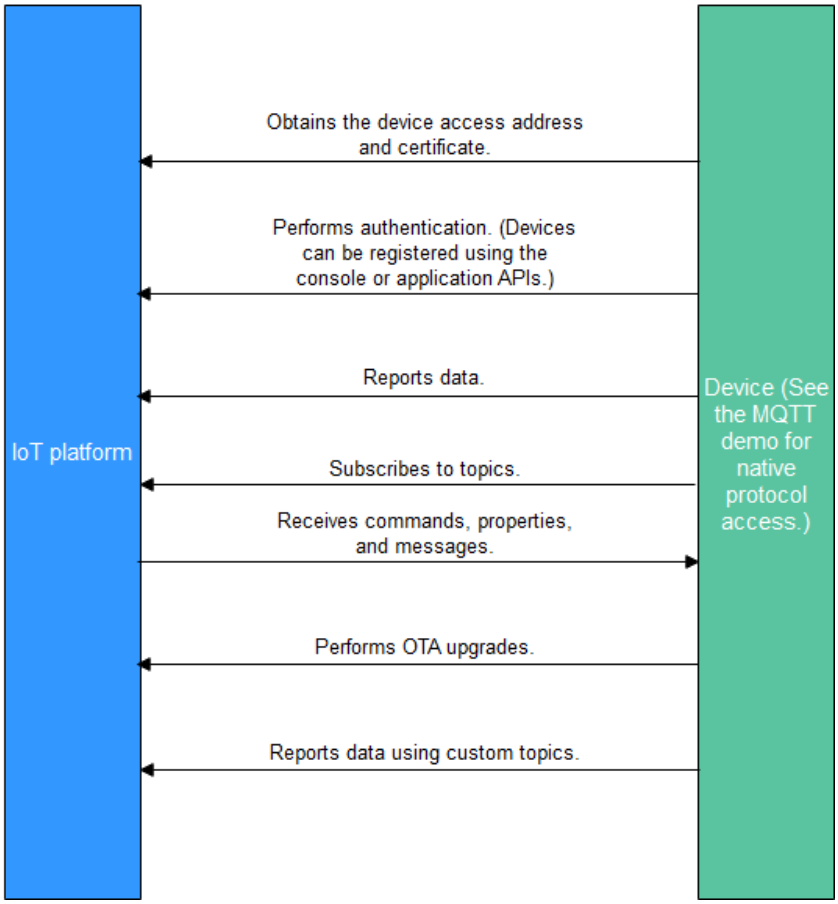
MQTTS is a variant of MQTT that uses TLS encryption. MQTTS devices communicate with the platform using encrypted data transmission.



Service Flow

MQTT devices communicate with the platform without data encryption. For security purposes, MQTTS access is recommended.

You are advised to use the **IoT Device SDK** to connect devices to the platform over MQTTS.



1. Create a product on the IoTDA console or by calling the API **Creating a Product**.
2. Register a device on the **IoTDA console** or calling the API **Creating a Device**.
3. The registered device can report messages and properties, receive commands, properties, and messages, perform OTA upgrades, and report data using custom topics. For details about preset topics of the platform, see **Topic Definition**.

NOTE

You can use MQTT.fx to debug access using the native MQTT protocol. For details, see **Developing an MQTT-based Smart Street Light Online**.

Constraints

Description	Constraint
Number of concurrent connections to a directly connected MQTT device	1
Connection setup requests of an account per second on the device side	• Basic edition: 100 • Standard edition: See **Specifications**.

Description	Constraint
Number of upstream requests for an instance per second on the device side (when average message payload is 512 bytes)	• Basic edition: 500 • Standard edition: See **Specifications**.
Number of upstream messages for an MQTT connection	50 per second
Bandwidth of an MQTT connection (upstream messages)	1 MB (default)
Length of a publish message sent over an MQTT connection (Oversized messages will be rejected.)	1 MB
Standard MQTT protocol	MQTT v5.0, MQTT v3.1.1, and MQTT v3.1
Differences from the standard MQTT protocol	• Not supported: QoS 2 • Not supported: **will** and **retain msg**
Security levels supported by MQTT	TCP channel and TLS protocols (TLS v1, TLS v1.1, TLS v1.2, and TLS v1.3)
Recommended heartbeat interval for MQTT connections	Range: 30s to 1200s; recommended: 120s
MQTT message publish and subscription	A device can only publish and subscribe to messages of its own topics.
Number of subscriptions for an MQTT connection	100
Length of a custom MQTT topic	128 bytes
Number of custom MQTT topics added to a product	10
Number of CA certificates uploaded for an account on the device side	100

Communication Between MQTT Devices and the Platform

The platform communicates with MQTT devices through topics, and they exchange messages, properties, and commands using preset topics. You can also create custom topics for connected devices to meet specific requirements.

Data Type	Message Type	Description
Upstream data	Reporting device properties	Devices report property data in the format defined in the product model.
	Reporting device messages	If a device cannot report data in the format defined in the product model, the device can report data to the platform using the device message reporting API. The platform forwards the messages reported by devices to an application or other Huawei Cloud services for storage and processing.
	Gateway reporting device properties in batches	A gateway reports property data of multiple devices to the platform.
	Reporting device events	Devices report event data in the format defined in the product model.
Downstream data	Delivering platform messages	The platform delivers data in a custom format to devices.
	Setting device properties	A product model defines the properties that the platform can configure for devices. The platform or application can modify the properties of a specific device.
	Querying device properties	The platform or application can query real-time property data of a specific device.
	Delivering platform commands	The platform or application delivers commands in the format defined in the product model to devices.
	Delivering platform events	The platform or application delivers events in the format defined in the product model to devices.

Preset Topics

The following table lists the preset topics of the platform.

Category	Function	Topic	Publisher	Subscriber
Device message related topics	**Device Reporting a Message**	$oc/devices/{device_id}/sys/messages/up	Device	Platform
	Platform Delivering a Message	$oc/devices/{device_id}/sys/messages/down	Platform	Device
Device command related topics	**Platform Delivering a Command**	$oc/devices/{device_id}/sys/commands/request_id={request_id}	Platform	Device
	Device Returning a Command Response	$oc/devices/{device_id}/sys/commands/response/request_id={request_id}	Device	Platform
Device property related topics	**Device Reporting Properties**	$oc/devices/{device_id}/sys/properties/report	Device	Platform
	Reporting Property Data by a Gateway	$oc/devices/{device_id}/sys/gateway/sub_devices/properties/report	Device	Platform
	Setting Device Properties	$oc/devices/{device_id}/sys/properties/set/request_id={request_id}	Platform	Device
	Returning a Response to Property Settings	$oc/devices/{device_id}/sys/properties/set/response/request_id={request_id}	Device	Platform
	Querying Device Properties	$oc/devices/{device_id}/sys/properties/get/request_id={request_id}	Platform	Device

Category	Function	Topic	Publisher	Subscriber
	Device Returning a Response for a Property Query The response does not affect device properties and shadows.	$oc/devices/{device_id}/sys/properties/get/response/request_id={request_id}	Device	Platform
	Obtaining Device Shadow Data from the Platform	$oc/devices/{device_id}/sys/shadow/get/request_id={request_id}	Device	Platform
	Returning a Response to a Request for Obtaining Device Shadow Data	$oc/devices/{device_id}/sys/shadow/get/response/request_id={request_id}	Platform	Device
Device event related topics	**Reporting a Device Event**	$oc/devices/{device_id}/sys/events/up	Device	Platform
	Delivering an Event	$oc/devices/{device_id}/sys/events/down	Platform	Device

You can create custom topics on the console to report personalized data. For details, see **Custom Topic Communications**.

TLS Support for MQTT

TLS is recommended for secure transmission between devices and the platform. Currently, TLS v1.1, v1.2, v1.3, and GMTLS are supported. TLS v1.3 is recommended. TLS v1.1 will not be supported in the future. GMTLS is supported only by the enterprise edition using Chinese cryptographic algorithms.

When TLS connections are used for the basic edition, standard edition, and enterprise edition that support general cryptographic algorithms, the IoT platform supports the following cipher suites:

- TLS_AES_256_GCM_SHA384
- TLS_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA

When the enterprise edition that supports Chinese cryptographic algorithms uses TLS connections, the IoT platform supports the following cipher suites:

- ECC_SM4_GCM_SM3
- ECC_SM4_CBC_SM3
- ECDHE_SM4_GCM_SM3
- ECDHE_SM4_CBC_SM3
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

NOTE

CBC cipher suites may pose security risks.

FAQ

MQTT-based Device Access

3.8.2 Java Demo Usage Guide

Overview

This topic uses Java as an example to describe how to connect a device to the platform over MQTTS or MQTT and how to use **platform APIs** to report properties and subscribe to a topic for receiving commands.

NOTE

The code snippets in this document are only examples and are for trial use only. To put them into commercial use, obtain the IoT Device SDKs of the corresponding language for integration by referring to **Obtaining Resources**.

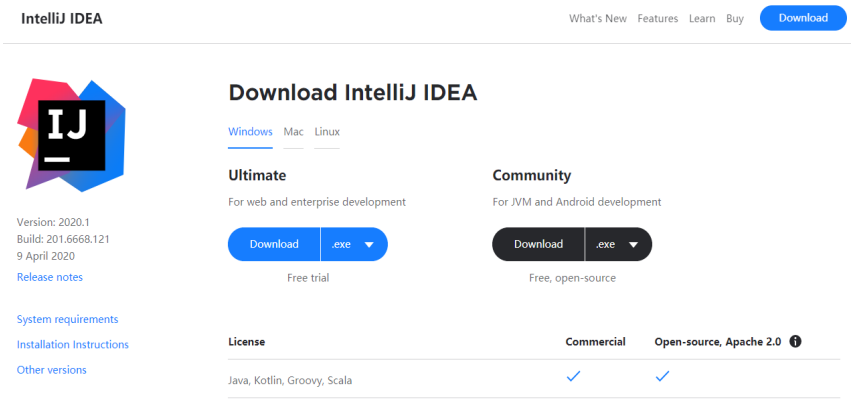
Prerequisites

- You have obtained the device access address from the **IoTDA console**. For details about how to obtain the address, see **Platform Connection Information**.
- You have created a product and a device on the **IoTDA console**. For details, see **Creating a Product**, **Registering an Individual Device**, and **Registering a Batch of Devices**.

Preparations

Installing IntelliJ IDEA

1. Go to the **IntelliJ IDEA website** to download and install a desired version. The following uses Windows 64-bit IntelliJ IDEA 2019.2.3 Ultimate as an example.

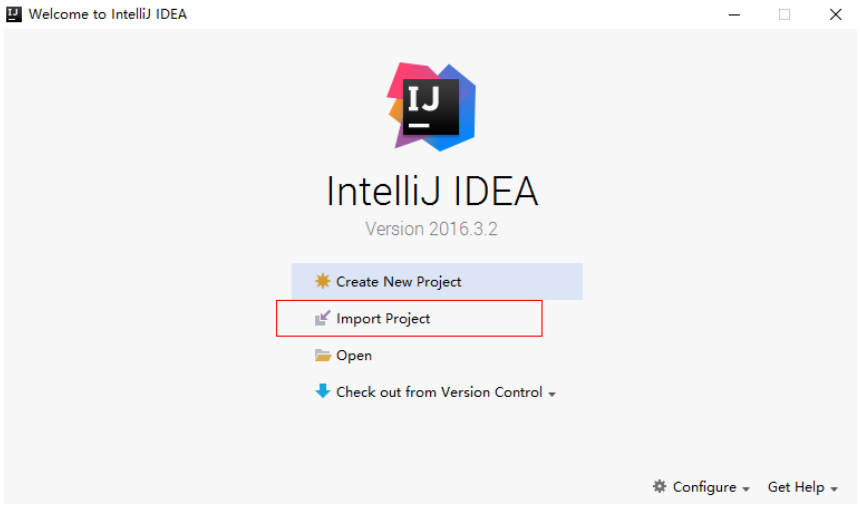


2. After the download is complete, run the installation file and install IntelliJ IDEA as prompted.

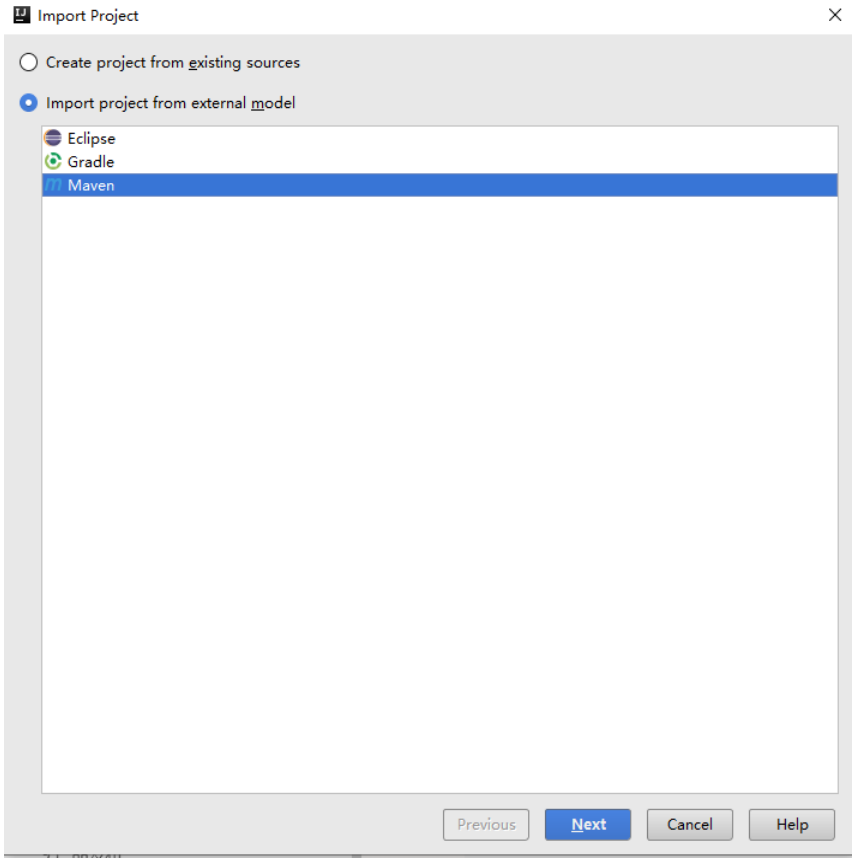
Importing Sample Code

Step 1 Download the **Java demo**.

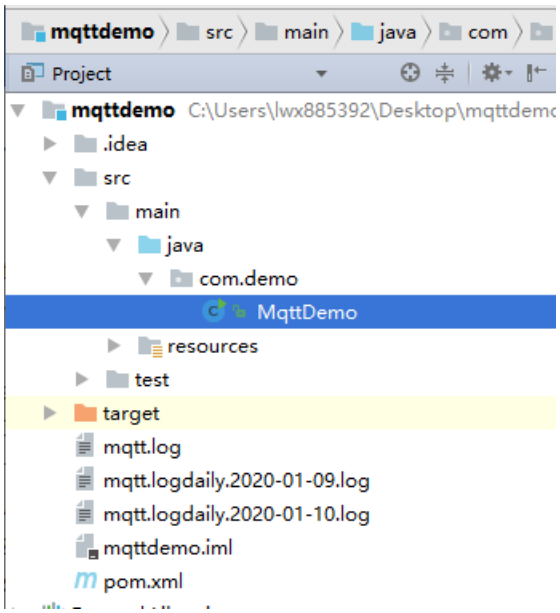
Step 2 Open the IDEA developer tool and click **Import Project**.



Step 3 Select the downloaded Java demo and click **Next**.



Step 4 Import the sample code.



----End

Establishing a Connection

To connect a device or gateway to the platform, upload the device information to bind the device or gateway to the platform.

1. Before establishing a connection, modify the following parameters:

```
// MQTT connection address of the platform (Replace it with the domain name of the IoT platform that the device is connected to.)
static String serverIp = "xxx.myhuaweicloud.com";
// Device ID and secret obtained during device registration (Replace them with the actual values.)
static String deviceId = "722cb*****************";
static String secret = "******";
```

   - **serverIp** indicates the device connection address of the platform. To obtain this address, see **Platform Connection Information**. (After obtaining the domain name, run the **ping** *Domain name* command in the CLI to obtain the corresponding IP address.)
   - **deviceId** and **secret** indicate the device ID and secret, which can be obtained after **the device is registered**.

2. Use MqttClient to set up a connection. The recommended heartbeat interval for MQTT connections is 120 seconds. For details, see **Constraints**.

```
MqttConnectOptions options = new MqttConnectOptions();
options.setCleanSession(false);
options.setKeepAliveInterval(120); // Set the heartbeat interval from 30 to 1200 seconds.
options.setConnectionTimeout(5000);
options.setAutomaticReconnect(true);
options.setUserName(deviceId);
options.setPassword(getPassword().toCharArray());
client = new MqttAsyncClient(url, getClientId(), new MemoryPersistence());
client.setCallback(callback);
```

   Port 1883 is a non-encrypted MQTT access port, and port 8883 is an encrypted MQTTS access port (that uses SSL to load a certificate).

```
if (isSSL) {
    url = "ssl://" + serverIp + ":" + 8883; // MQTTS connection
} else {
    url = "tcp://" + serverIp + ":" + 1883; // MQTT connection
}
```

   To establish an MQTTS connection, load the SSL certificate of the server and add the **SocketFactory** parameter. The **DigiCertGlobalRootCA.jks** file is stored in the **resources** directory of the demo. It is used by the device to verify the platform identity when the device connects to the platform. You can download the certificate file using the link provided in **Certificates**.

```
options.setSocketFactory(getOptionSocketFactory(MqttDemo.class.getClassLoader().getResource("DigiCertGlobalRootCA.jks").getPath()));
```

3. Call **client.connect(options, null, new IMqttActionListener())** to initiate a connection. The **MqttConnectOptions** parameter is passed.

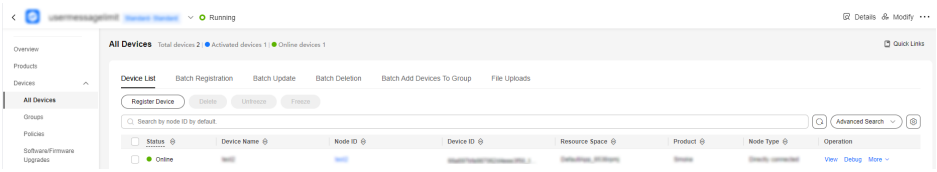
```
client.connect(options, null, new IMqttActionListener()
```

4. The password passed by calling **options.setPassword()** is encrypted during creation of **MqttConnectOptions**. **getPassword()** is used to obtain the encrypted password.

```
public static String getPassword() {
    return sha256_mac(secret, getTimeStamp());
}
/* Call the SHA-256 algorithm for hash calculation. */
public static String sha256_mac(String message, String tStamp) {
    String passWord = null;
    try {
        Mac sha256_HMAC = Mac.getInstance("HmacSHA256");
        SecretKeySpec secret_key = new SecretKeySpec(tStamp.getBytes(), "HmacSHA256");
        sha256_HMAC.init(secret_key);byte[] bytes = sha256_HMAC.doFinal(message.getBytes());
        passWord = byteArrayToHexString(bytes);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return passWord;
```


5. After the connection is established, the device becomes online.

Figure 3-206 Device list - Device online status



If the connection fails, the onFailure function executes backoff reconnection. The example code is as follows:

```
@Override
public void onFailure(IMqttToken iMqttToken, Throwable throwable) {
    System.out.println("Mqtt connect fail.");

    // Backoff reconnection
    int lowBound = (int) (defaultBackoff * 0.8);
    int highBound = (int) (defaultBackoff * 1.2);
    long randomBackOff = random.nextInt(highBound - lowBound);
    long backOffWithJitter = (int) (Math.pow(2.0, (double) retryTimes)) * (randomBackOff +
lowBound);
    long waitTImeUntilNextRetry = (int) (minBackoff + backOffWithJitter) > maxBackoff ?
maxBackoff : (minBackoff + backOffWithJitter);
    System.out.println("----  " + waitTImeUntilNextRetry);
    try {
        Thread.sleep(waitTImeUntilNextRetry);
        } catch (InterruptedException e) {
            System.out.println("sleep failed, the reason is" + e.getMessage().toString());
        }
        retryTimes++;
        MqttDemo.this.connect(true);
}
```

Subscribing to a Topic for Receiving Commands

Only devices that subscribe to a specific topic can receive messages about the topic published by the broker. For details on the preset topics, see **Topics**. For details about the API, see **Platform Delivering a Command**.

```
// Subscribe to a topic for receiving commands.
client.subscribe(getCmdRequestTopic(), qosLevel, null, new IMqttActionListener();
```

getCmdRequestTopic() is used to obtain the topic for receiving commands from the platform and subscribe to the topic.

```
public static String getCmdRequestTopic() {
    return "$oc/devices/" + deviceId + "/sys/commands/#";
}
```

Reporting Properties

Devices can report their properties to the platform. For details, see **Reporting Device Properties**.

```
// Report JSON data. service_id must be the same as that defined in the product model.
String jsonMsg = "{\"services\": [{\"service_id\": \"Temperature\",\"properties\": {\"value\": 57}},{\"service_id
\": \"Battery\",\"properties\": {\"level\": 80}}]}";
MqttMessage message = new MqttMessage(jsonMsg.getBytes());
client.publish(getRreportTopic(), message, qosLevel, new IMqttActionListener();
```

The message body **jsonMsg** is assembled in JSON format, and **service_id** must be the same as that defined in the product model. **properties** indicates a device

property, and **57** indicates the property value. **event_time** indicates the UTC time when the device reports data. If this parameter is not specified, the system time is used by default.

After a device or gateway is connected to the platform, you can call **MqttClient.publish(String topic,MqttMessage message)** to report device properties to the platform.

getRreportTopic() is used to obtain the topic for reporting data.

```
public static String getRreportTopic() {
    return "$oc/devices/" + deviceId + "/sys/properties/report";
}
```

Viewing Reported Data

After the **main** method is called, you can view the reported device property data on the device details page. For details about the API, see **Device Reporting Properties**.

Figure 3-207 Viewing reported data - level

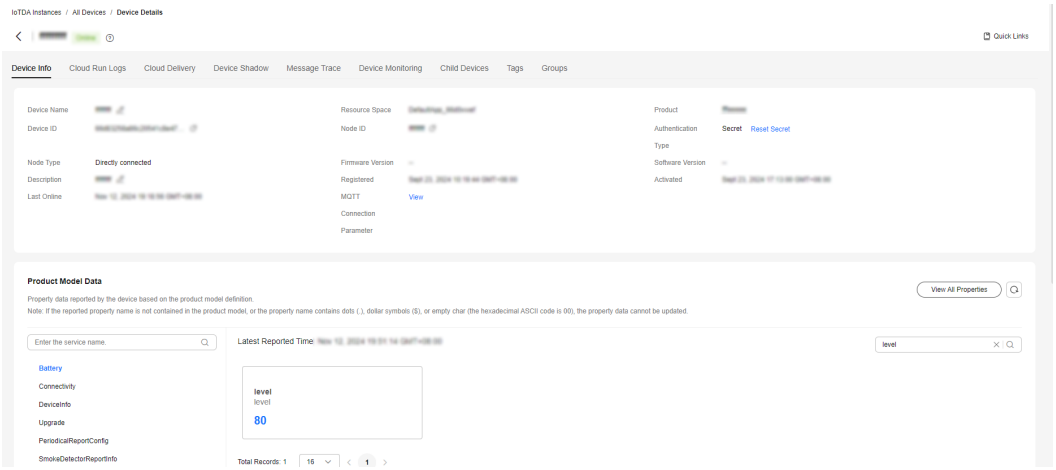
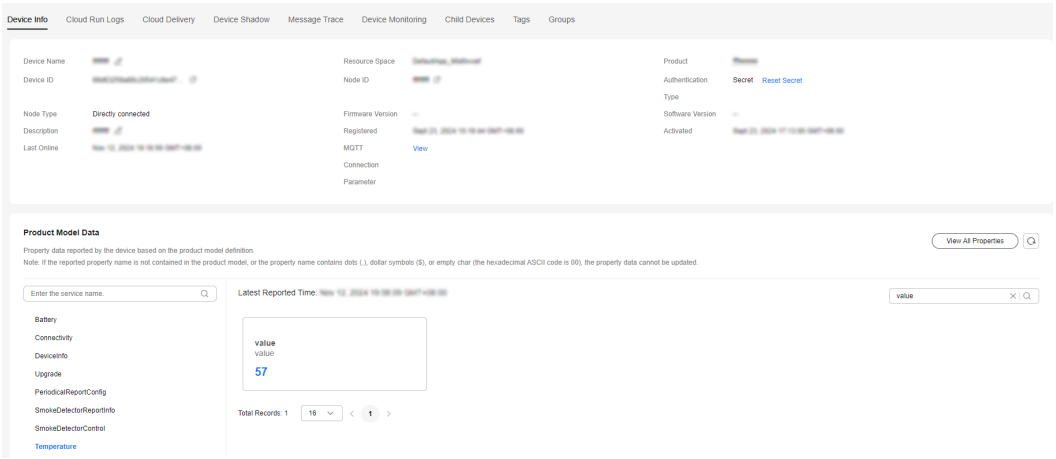


Figure 3-208 Viewing reported data - temperature_value



NOTE

If the latest data is not displayed on the device details page, check whether the services and properties reported by the device are the same as those in the product model.

Related Resources

You can refer to the **MQTT or MQTTS API Reference on the Device Side** to connect MQTT devices to the platform. You can also **develop an MQTT-based smart street light online** to quickly verify whether they can interact with the IoT platform to publish or subscribe to messages.

NOTE

Synchronous commands require device responses. For details, see **Upstream Response Parameters**.

3.8.3 Python Demo Usage Guide

Overview

This topic uses Python as an example to describe how to connect a device to the platform over MQTTS or MQTT and how to use **platform APIs** to report properties and subscribe to a topic for receiving commands.

NOTE

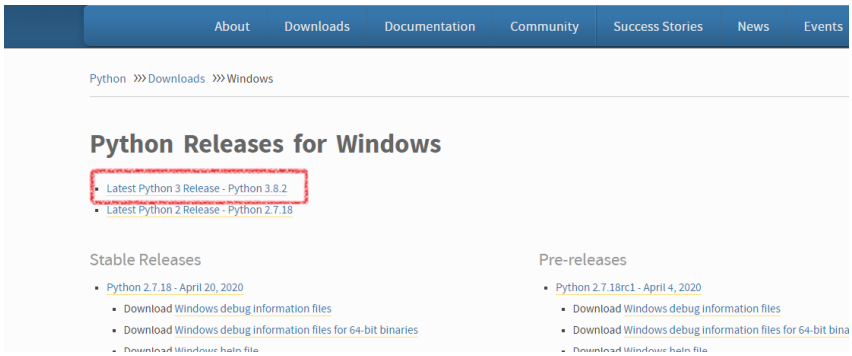
The code snippets in this document are only examples and are for trial use only. To put them into commercial use, obtain the IoT Device SDKs of the corresponding language for integration by referring to **Obtaining Resources**.

Prerequisites

- You have installed Python by following the instructions provided in **Installing Python**.
- You have installed a development tool (for example, PyCharm) by following the instructions provided in **Installing PyCharm**.
- You have obtained the device access address from the **IoTDA console**. For details about how to obtain the address, see **Platform Connection Information**.
- You have created a product and a device on the **IoTDA console**. For details, see **Creating a Product**, **Registering an Individual Device**, and **Registering a Batch of Devices**.

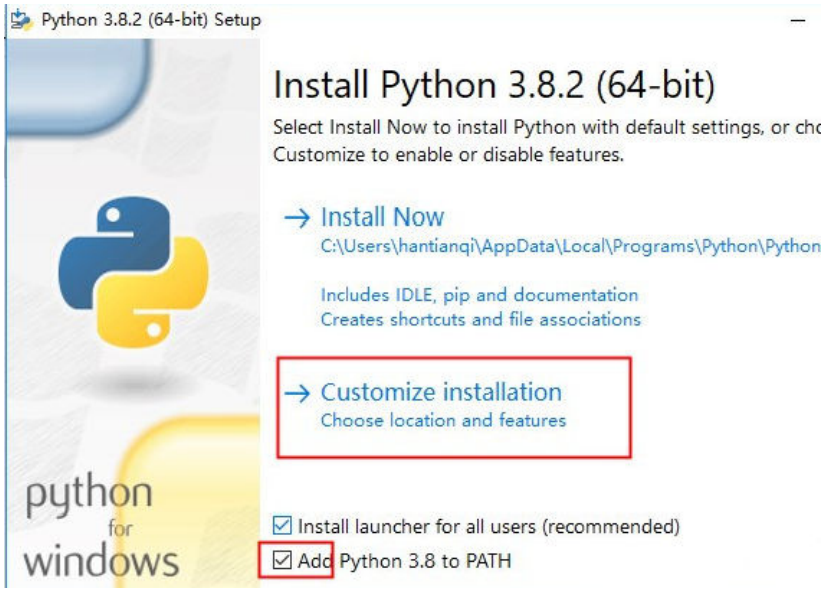
Preparations

- Installing Python
  a. Go to the **Python website** to download and install a desired version. (The following uses Windows OS as an example to describe how to install Python 3.8.2.)



b. After the download is complete, run the .exe file to install Python.

c. Select **Add python 3.8 to PATH** (if it is not selected, you need to manually configure environment variables), click **Customize installation**, and install Python as prompted.

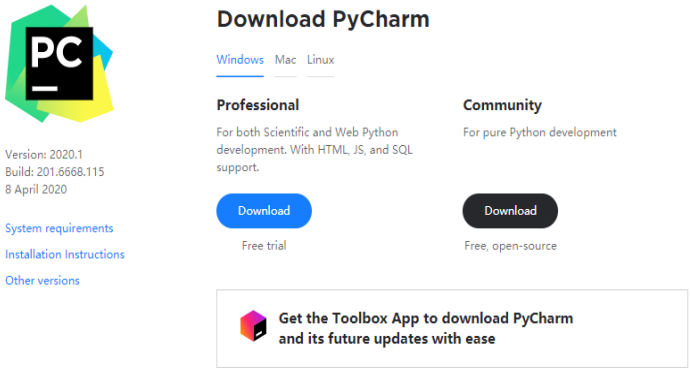


d. Check whether Python is installed.

Press **Win**+**R**, enter **cmd**, and press **Enter** to open the CLI. In the CLI, enter **python –V** and press **Enter**. If the Python version is displayed, the installation is successful.

- Installing PyCharm (If you have already installed PyCharm, skip this step.)

a. Visit the **PyCharm website**, select a version, and click **Download**.



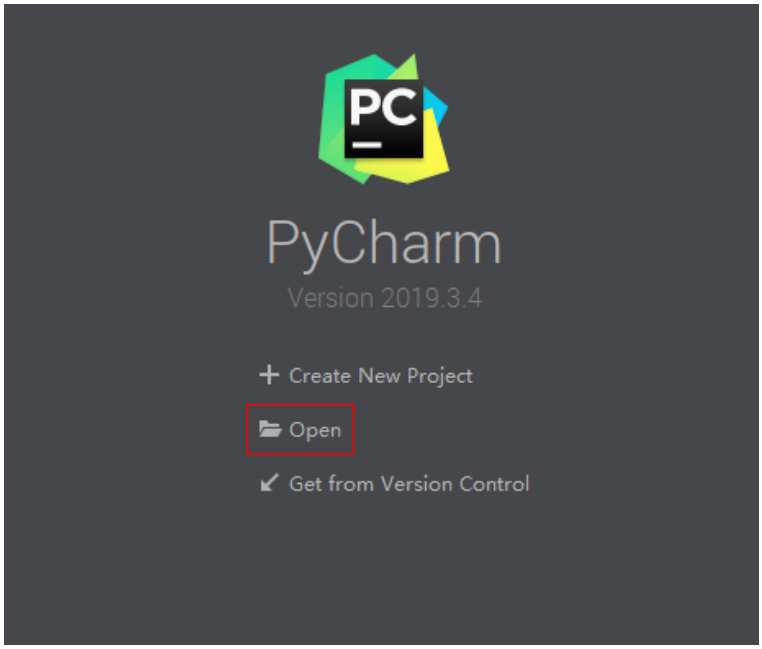
The professional edition is recommended.

b. Run the .exe file and install PyCharm as prompted.

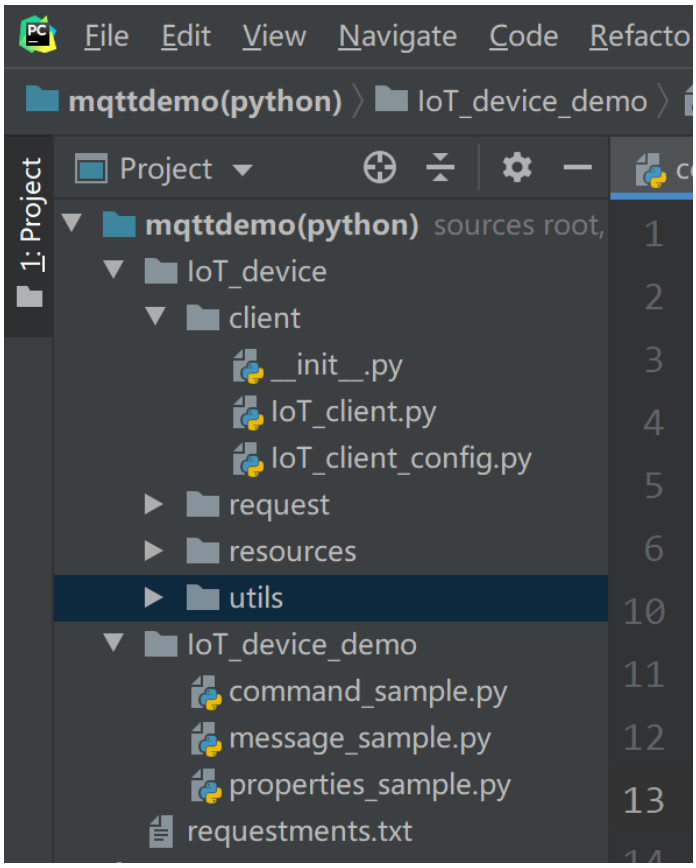
Importing Sample Code

Step 1 Download the **QuickStart (Python)**.

Step 2 Run PyCharm, click **Open**, and select the sample code downloaded.



Step 3 Import the sample code.



Description of the directories:

- **IoT_device_demo**: MQTT demo files

  message_sample.py: Demo for devices to send and receive messages

  command_sample.py: Demo for devices to respond to commands delivered by the platform

  properties_sample.py: Demo for devices to report properties
- **IoT_device/client**: Used for paho-mqtt encapsulation.

  IoT_client_config.py: client configurations, such as the device ID and secret

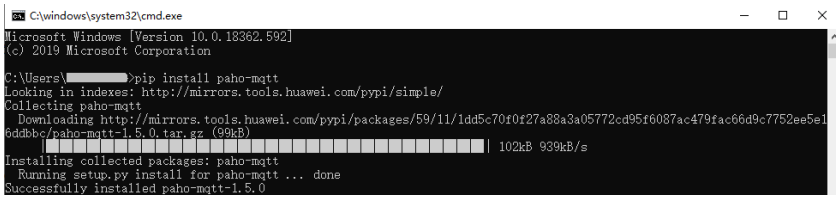
  IoT_client.py: MQTT-related function configurations, such as connection, subscription, publish, and response
- **IoT_device/Utils**: utility methods, such as those for obtaining the timestamp and encrypting a secret
- **IoT_device/resources**: Stores certificates.

  DigiCertGlobalRootCA.crt.pem is used by the device to verify the platform identity when the device connects to the platform. You can download the certificate file using the link provided in **Certificates**.
- **IoT_device/request**: Encapsulates device properties, such as commands, messages, and properties.

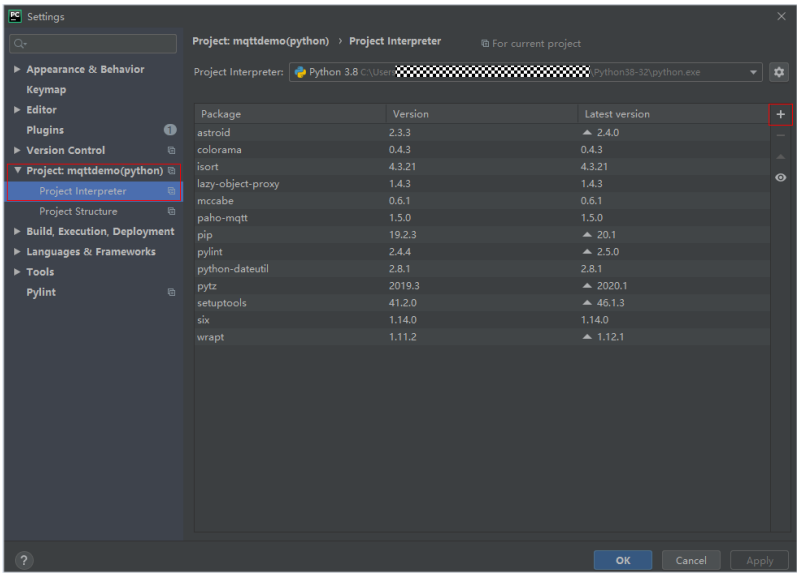
Step 4 (Optional) Install the paho-mqtt library, which is a third-party library that uses the MQTT protocol in Python. If the paho-mqtt library has already been installed, skip this step. You can install paho-mqtt using either of the following methods:

- Method 1: Use the pip tool to install paho-mqtt in the CLI. (The tool is already provided when installing Python.)

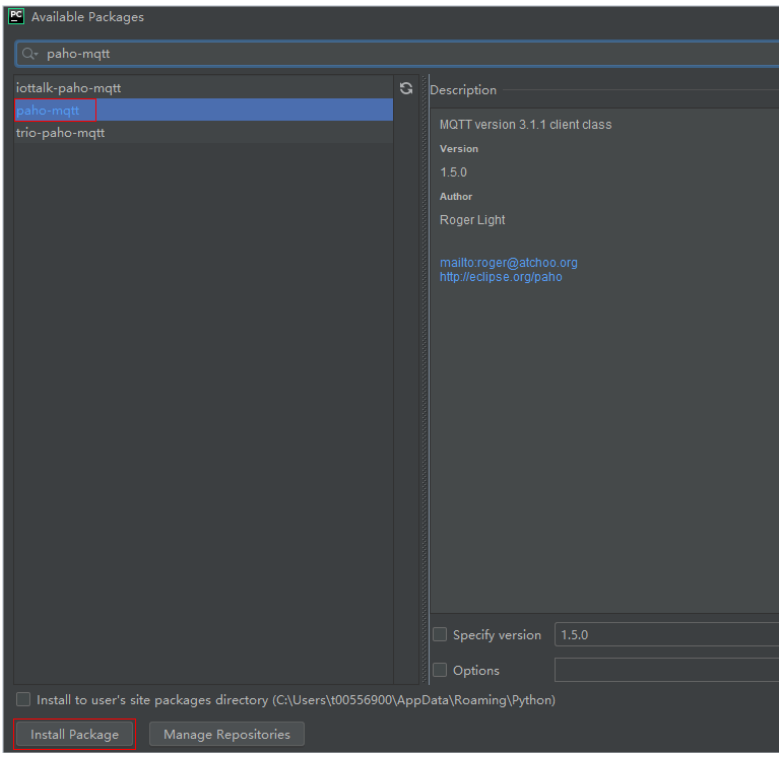
In the CLI, enter **pip install paho-mqtt** and press **Enter**. If the message **Successfully installed paho-mqtt** is displayed, the installation is successful. If a message is displayed indicating that the pip command is not an internal or external command, check the Python environment variables. See the figure below.



- Method 2: Install paho-mqtt using PyCharm.
  a. Open PyCharm, choose **File** > **Settings** > **Project Interpreter**, and click the plus icon (+) on the right side to search for **paho-mqtt**.



  b. Click **Install Package** in the lower left corner.



----End

Establishing a Connection

To connect a device or gateway to the platform, upload the device information to bind the device or gateway to the platform.

1. Before establishing a connection, modify the following parameters. The **IoTClientConfig** class is used to configure client information.

```
# Client configurations
client_cfg = IoTClientConfig(server_ip='iot-mqtts.cn-north-4.myhuaweicloud.com',
device_id='5e85a55f60b7b804c51ce15c_py123', secret='******', is_ssl=True)
# Create a device.
iot_client = IotClient(client_cfg)
```

   - **server_ip** indicates the device connection address of the platform. To obtain this address, see **Platform Connection Information**. (After obtaining the domain name, run the **ping** *Domain name* command in the CLI to obtain the corresponding IP address.)
   - **device_id** and **secret** are returned after **the device is registered**.
   - **is_ssl**: **True** means to establish an MQTTS connection and **False** means to establish an MQTT connection.

2. Call the **connect** method to initiate a connection.

```
iot_client.connect()
```

   If the connection is successful, the following information is displayed:

```
----------------Connection successful !!!
```

   If the connection fails, the retreat_reconnection function executes backoff reconnection. The example code is as follows:

```
# Backoff reconnection
def retreat_reconnection(self):
    print("---- Backoff reconnection")
```



```
global retryTimes
minBackoff = 1
maxBackoff = 30
defaultBackoff = 1
low_bound = (int)(defaultBackoff * 0.8)
high_bound = (int)(defaultBackoff * 1.2)
random_backoff = random.randint(0, high_bound - low_bound)
backoff_with_jitter = math.pow(2.0, retryTimes) * (random_backoff + low_bound)
wait_time_until_next_retry = min(minBackoff + backoff_with_jitter, maxBackoff)
print("the next retry time is ", wait_time_until_next_retry, " seconds")
retryTimes += 1
time.sleep(wait_time_until_next_retry)
self.connect()
```

Subscribing to a Topic

Only devices that subscribe to a specific topic can receive messages about the topic published by the broker. For details on the preset topics, see **Topics**.

The **message_sample.py** file provides functions such as subscribing to topics, unsubscribing from topics, and reporting device messages.

To subscribe to a topic for receiving commands, do as follows:

```
iot_client.subscribe(r'$oc/devices/' + str(self.__device_id) + r'/sys/commands/#')
```

If the subscription is successful, information similar to the following is displayed. (**topic** indicates a custom topic, for example, **Topic_1**.)

```
------You have subscribed:  topic
```

Responding to a Command

The **command_sample.py** file provides the function of responding to commands delivered by the platform. For details about the API, see **Platform Delivering a Command**.

```
# Responding to commands delivered by the platform
def command_callback(request_id, command):
    # If the value of result_code is 0, the command is delivered . If the value is 1, the command fails to be delivered.
    iot_client.respond_command(request_id, result_code=0)
iot_client.set_command_callback(command_callback)
```

Reporting Properties

Devices can report their properties to the platform. For details about the API, see **Device Reporting Properties**.

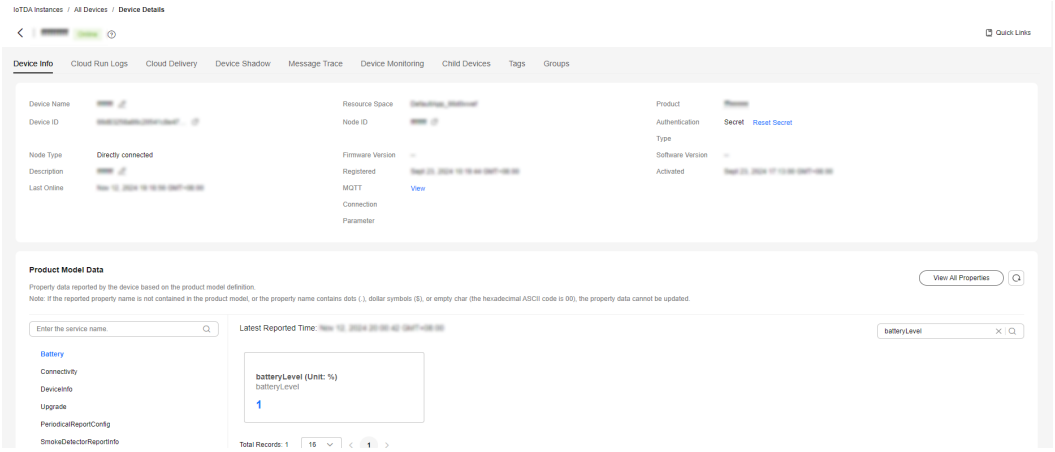
The **properties_sample.py** file provides the functions of reporting device properties, responding to platform settings, and querying device properties.

In the following code, the device reports properties to the platform every 10 seconds. **service_property** indicates a device property object. For details, see the **services_properties.py** file.

```
# Reporting properties periodically
while True:
    # Set properties based on the product model.
    service_property = ServicesProperties()
    service_property.add_service_property(service_id="Battery", property='batteryLevel', value=1)
    iot_client.report_properties(service_properties=service_property.service_property, qos=1)
    time.sleep(10)
```


If the reporting is successful, the reported device properties are displayed on the device details page.

Figure 3-209 Viewing reported data - Battery_batteryLevel



NOTE

If the latest data is not displayed on the device details page, check whether the services and properties reported by the device are the same as those in the product model.

Reporting a Message

Message reporting is the process in which a device reports messages to the platform. The **message_sample.py** file provides the message reporting function.

```
# Sending a message to the platform using the default topic
iot_client.publish_message('raw message: Hello Huawei cloud IoT')
```

If the message is reported, the following information is displayed:

```
Publish success---mid = 1
```

NOTE

Synchronous commands require device responses. For details, see **Upstream Response Parameters**.

3.8.4 Android Demo Usage Guide

Overview

This topic uses Android as an example to describe how to connect a device to the platform over MQTTS or MQTT and how to use **platform APIs** to report properties and subscribe to a topic for receiving commands.

NOTE

The code snippets in this document are only examples and are for trial use only. To put them into commercial use, obtain the IoT Device SDKs of the corresponding language for integration by referring to **Obtaining Resources**.

Prerequisites

- You have installed Android Studio. If not, install Android Studio by following the instructions provided on the **Android Studio website** and then install **the JDK**.
- You have obtained the device access address from the **IoTDA console**. For details about how to obtain the address, see **Platform Connection Information**.
- You have created a product and a device on the **IoTDA console**. For details, see **Creating a Product**, **Registering an Individual Device**, and **Registering a Batch of Devices**.

Preparations

- Install Android Studio.

  Go to the **Android Studio website** to download and install a desired version. The following uses Android Studio 3.5 running on 64-bit Windows as an example.

  Android Studio downloads

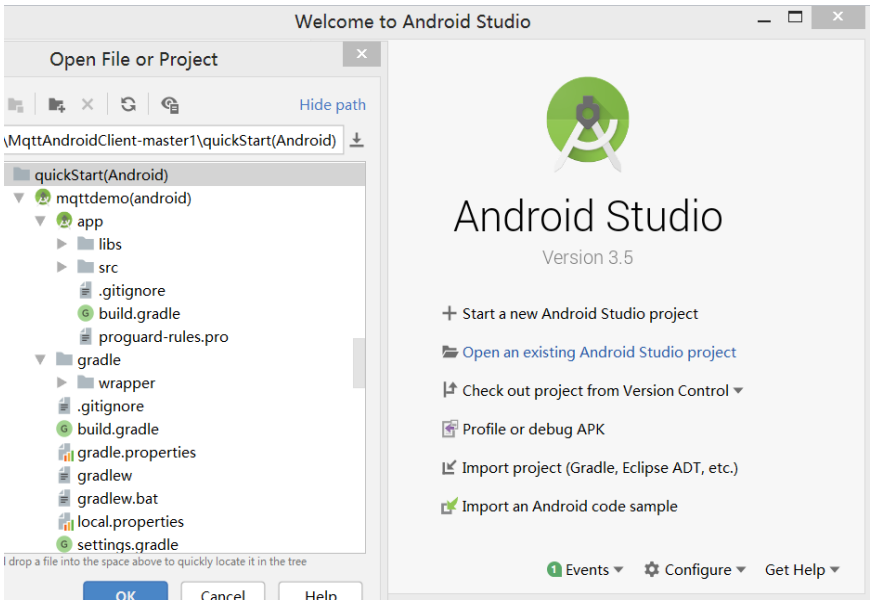
Platform	Android Studio package	Size	SHA-256 checksum
Windows (64-bit)	android-studio-ide-192.6392135-windows.exe Recommended	756 MB	07b6df807fda59e69f05b85ff6f6bd0c70d09e57fb151197155ef5f115f96e59
	android-studio-ide-192.6392135-windows.zip No .exe installer	770 MB	24f8f9ce467b935c25d89b90cad402d21dd45d4ba9af1ad35baeeb414609e483
Windows (32-bit)	android-studio-ide-192.6392135-windows32.zip No .exe installer	770 MB	7b24742726bbc8b40a55dab1f7cdff923ba384b233c21d35d6e96fa36320d067
Mac (64-bit)	android-studio-ide-192.6392135-mac.dmg	768 MB	c5dd347469be0d995e6b4d74ea72b3a6f2572e72b4eac37a0834b0a0984d9583
Linux (64-bit)	android-studio-ide-192.6392135-linux.tar.gz	772 MB	33ec9f61b20b71ca175cd39083b1379ebba896de78b826ea5df5d440c6adfd2a
Chrome OS	android-studio-ide-192.6392135-cros.deb	653 MB	59023aaabc7d5822fd7b1c5a71589b18e487ca8d7fd4320c3547ee0ad390e4ca

- Install the JDK. You can also use the built-in JDK of the IDE.
  a. Go to the **Oracle website** to download a desired version. The following uses JDK 8 for Windows x64 as an example.
  b. After the download is complete, run the installation file and install the JDK as prompted.

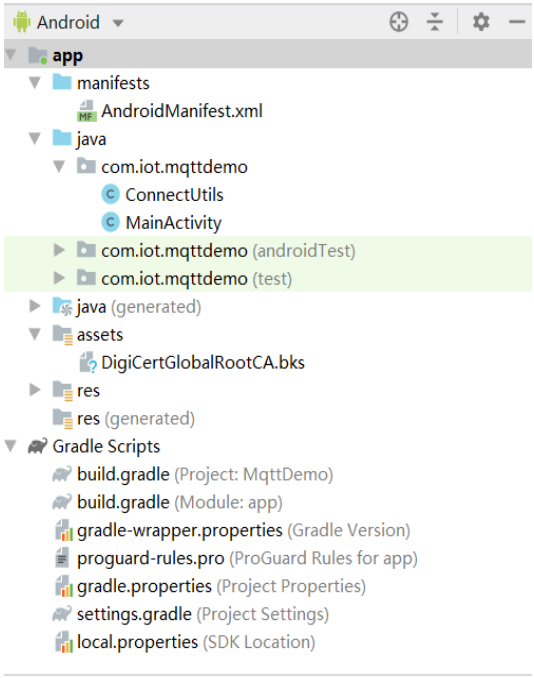
Importing Sample Code

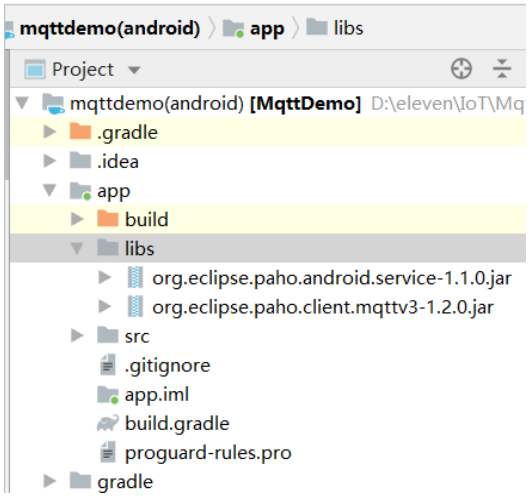
Step 1 Download the sample code **quickStart(Android)**.

Step 2 Run Android Studio, click **Open**, and select the sample code downloaded.



Step 3 Import the sample code.





Description of the directories:

- **manifests**: configuration file of the Android project
- **java**: Java code of the project

  MainActivity: demo UI class

  ConnectUtils: MQTT connection auxiliary class
- **asset**: native file of the project

  DigiCertGlobalRootCA.bks: certificate used by the device to verify the platform identity. It is used for login authentication when the device connects to the platform.
- **res**: project resource file (image, layout, and character string)
- **gradle**: global Gradle build script of the project
- **libs**: third-party JAR packages used in the project

  org.eclipse.paho.android.service-1.1.0.jar: component for Android to start the background service component to publish and subscribe to messages

  org.eclipse.paho.client.mqttv3-1.2.0.jar: MQTT java client component

Step 4 (Optional) Understand the key project configurations in the demo. (By default, you do not need to modify the configurations.)

- **AndroidManifest.xml**: Add the following information to support the MQTT service.

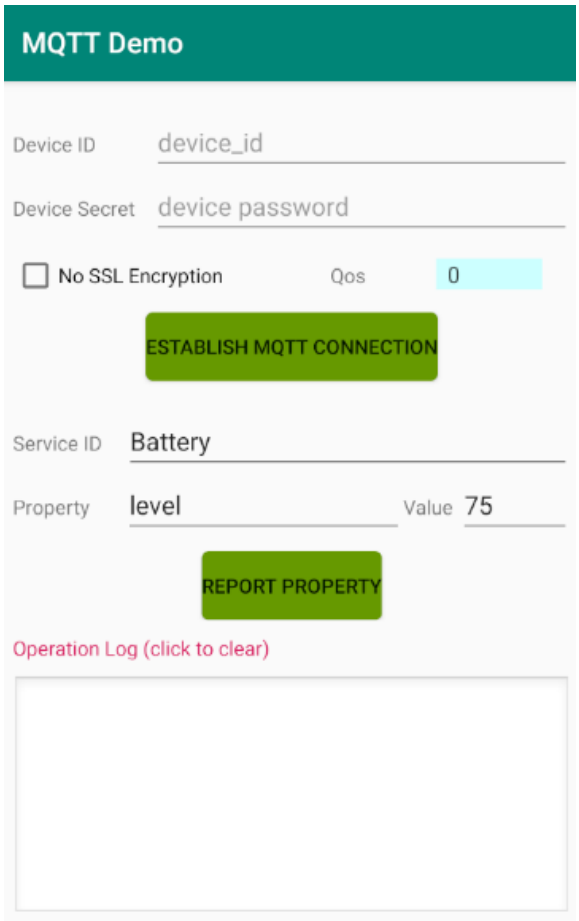
```
<service android:name="org.eclipse.paho.android.service.MqttService" />
```

- **build.gradle**: Add dependencies and import the JAR packages required for the two MQTT connections in the **libs** directory. (You can also add the JAR package to the website for reference.)

```
implementation files('libs/org.eclipse.paho.android.service-1.1.0.jar')
implementation files('libs/org.eclipse.paho.client.mqttv3-1.2.0.jar')
```

----End

UI Display



1. The **MainActivity** class provides UI display. Enter the device ID and secret, which are obtained after the device is registered on the IoTDA console or by calling the API **Creating a Device**.
2. In the example, the domain name accessed by the device is used by default. (The domain name must match and be used together with the corresponding **certificate file** during SSL-encrypted access.)
   ```
   private final static String IOT_PLATFORM_URL = "iot-mqtts.cn-north-4.myhuaweicloud.com";
   ```
3. Select SSL encryption or no encryption when establishing a connection on the device side and set the QoS mode to **0** or **1**. Currently, QoS 2 is not supported. For details, see **Constraints**.
   ```
   checkbox_mqtt_connet_ssl.setOnCheckedChangeListener(new
   CompoundButton.OnCheckedChangeListener() {
       @Override
       public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
           if (isChecked) {
               isSSL = true;
               checkbox_mqtt_connet_ssl.setText ("SSL encryption");
           } else {
               isSSL = false;
               checkbox_mqtt_connet_ssl.setText ("no SSL encryption");
           }
       }
   })
   ```


Establishing a Connection

To connect a device or gateway to the platform, upload the device information to bind the device or gateway to the platform.

1. Call the **MainActivity** class to establish an MQTT or MQTTS connection. By default, MQTT uses port 1883, and MQTTS uses port 8883 (a certificate must be loaded).

```
if (isSSL) {
    editText_mqtt_log.append("Starting to establish an MQTTS connection" + "\n");
    serverUrl = "ssl://" + IOT_PLATFORM_URL + ":8883";
} else {
    editText_mqtt_log.append("Starting to establish an MQTT connection" + "\n");
    serverUrl = "tcp://" + IOT_PLATFORM_URL + ":1883";
}
```

2. Call the **getMqttsCertificate** method in the **ConnectUtils** class to load an SSL certificate. This step is required only if an MQTTS connection is established.

   DigiCertGlobalRootCA.bks: certificate used by the device to verify the platform identity for login authentication when the device connects to the platform. You can download the certificate file using the link provided in **Certificates**.

```
SSLContext sslContext = SSLContext.getInstance("SSL");
KeyStore keyStore = KeyStore.getInstance("bks");
The keyStore.load(context.getAssets().open("DigiCertGlobalRootCA.bks"), null);// Load the certificate in the libs directory.
TrustManagerFactory trustManagerFactory = TrustManagerFactory.getInstance("X509");
trustManagerFactory.init(keyStore);
TrustManager[] trustManagers = trustManagerFactory.getTrustManagers();
sslContext.init(null, trustManagers, new SecureRandom());
sslSocketFactory = sslContext.getSocketFactory();
```

3. Call the **intitMqttConnectOptions** method in the **MainActivity** class to initialize MqttConnectOptions. The recommended heartbeat interval for MQTT connections is 120 seconds. For details, see **Constraints**.

```
mqttAndroidClient = new MqttAndroidClient(mContext, serverUrl, clientId);
private MqttConnectOptions intitMqttConnectOptions(String currentDate) {
    String password =
ConnectUtils.sha256_HMAC(editText_mqtt_device_connect_password.getText().toString(),
currentDate);
    MqttConnectOptions mqttConnectOptions = new MqttConnectOptions();
    mqttConnectOptions.setAutomaticReconnect(true);
    mqttConnectOptions.setCleanSession(true);
    mqttConnectOptions.setKeepAliveInterval(120);
    mqttConnectOptions.setConnectionTimeout(30);
    mqttConnectOptions.setUserName(editText_mqtt_device_connect_deviceId.getText().toString());
    mqttConnectOptions.setPassword(password.toCharArray());
    return mqttConnectOptions;
}
```

4. Call the **connect** method in the **MainActivity** class to set up a connection and the **setCallback** method to process the message returned after the connection is set up.

```
mqttAndroidClient.connect(mqttConnectOptions, null, new IMqttActionListener()
mqttAndroidClient.setCallback(new MqttCallBack4IoTHub());
```

If the connection fails, the onFailure function in initMqttConnects executes backoff reconnection. Sample code:

```
@Override
public void onFailure(IMqttToken asyncActionToken, Throwable exception) {
    exception.printStackTrace();
    Log.e(TAG, "Fail to connect to: " + exception.getMessage());
    editText_mqtt_log.append("Failed to set up the connection: "+ exception.getMessage() + "\n");
```



```
    // Backoff reconnection
    int lowBound = (int) (defaultBackoff * 0.8);
    int highBound = (int) (defaultBackoff * 1.2);
    long randomBackOff = random.nextInt(highBound - lowBound);
    long backOffWithJitter = (int) (Math.pow(2.0, (double) retryTimes)) * (randomBackOff + lowBound);
    long waitTImeUntilNextRetry = (int) (minBackoff + backOffWithJitter) > maxBackoff ? maxBackoff :
(minBackoff + backOffWithJitter);
    try {
        Thread.sleep(waitTImeUntilNextRetry);
    } catch (InterruptedException e) {
        System.out.println("sleep failed, the reason is" + e.getMessage().toString());
    }
    retryTimes++;
    MainActivity.this.initMqttConnects();
}
```

Subscribing to a Topic

Only devices that subscribe to a specific topic can receive messages about the topic published by the broker. For details on the preset topics, see **Topics**.

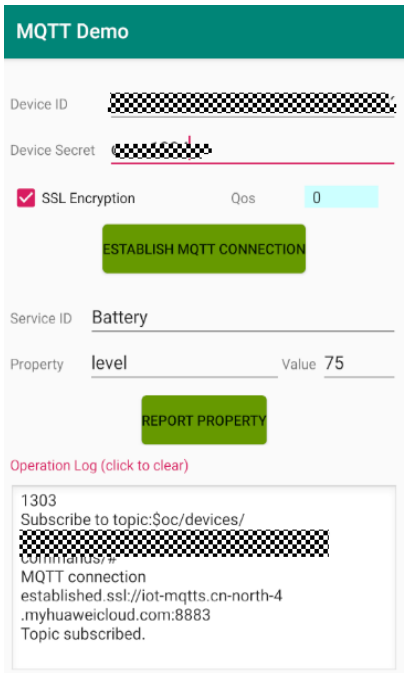
The **MainActivity** class provides the methods for delivering subscription commands to topics, subscribing to topics, and unsubscribing from topics.

```
String mqtt_sub_topic_command_json = String.format("$oc/devices/%s/sys/commands/#",
editText_mqtt_device_connect_deviceId.getText().toString());
mqttAndroidClient.subscribe(getSubscriptionTopic(), qos, null, new IMqttActionListener()
mqttAndroidClient.unsubscribe(getSubscriptionTopic(), null, new IMqttActionListener()
```

If the connection is established, you can subscribe to the topic using a callback function.

```
mqttAndroidClient.connect(mqttConnectOptions, null, new IMqttActionListener() {
    @Overridepublic void onSuccess(IMqttToken asyncActionToken) {
        ......
        subscribeToTopic();
}
```

After the connection is established, the following information is displayed in the log area of the application page:



Reporting Properties

Devices can report their properties to the platform. For details about the API, see **Device Reporting Properties**.

The **MainActivity** class implements the property reporting topic and property reporting.

```
String mqtt_report_topic_json = String.format("$oc/devices/%s/sys/properties/report",
editText_mqtt_device_connect_deviceId.getText().toString());
MqttMessage mqttMessage = new MqttMessage();
mqttMessage.setPayload(publishMessage.getBytes());
mqttAndroidClient.publish(publishTopic, mqttMessage);
```

If the reporting is successful, the reported device properties are displayed on the device details page.

Figure 3-210 Viewing reported data - PeriodicalReportConfig

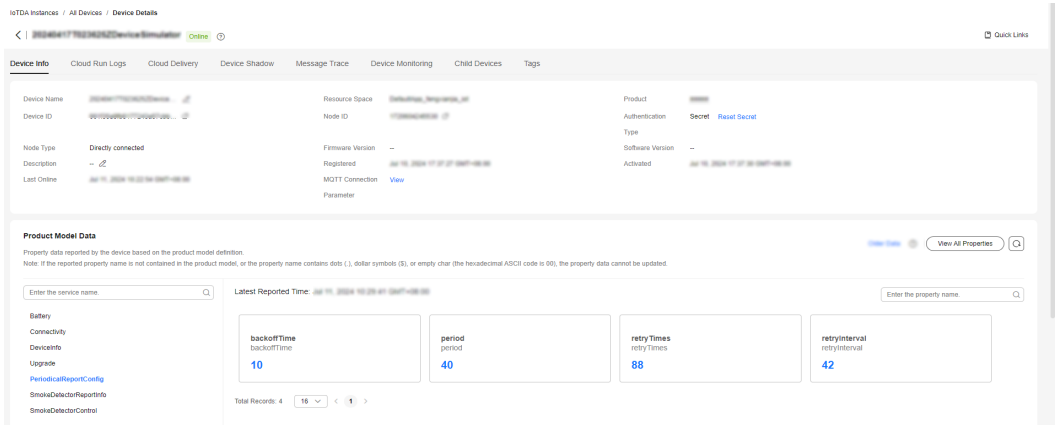
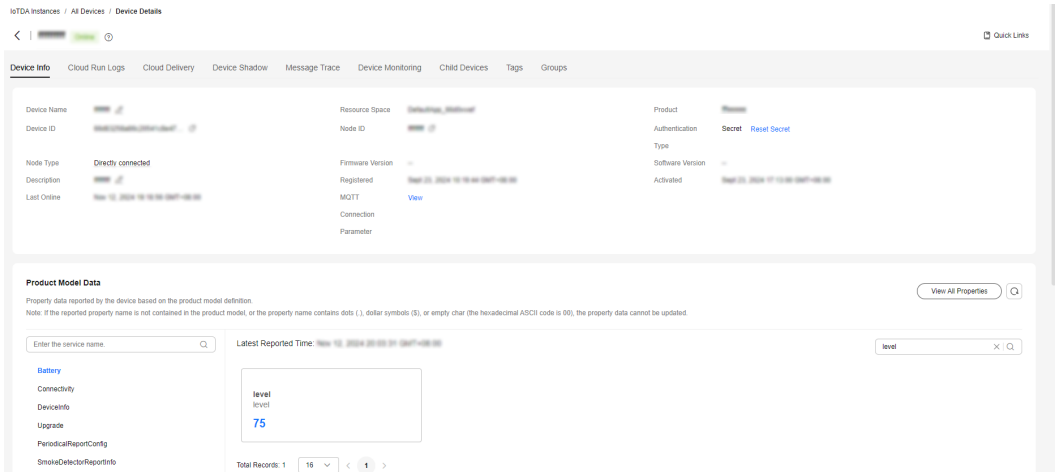


Figure 3-211 Viewing reported data - Battery_level



NOTE

If the latest data is not displayed on the device details page, check whether the services and properties reported by the device are the same as those in the product model.

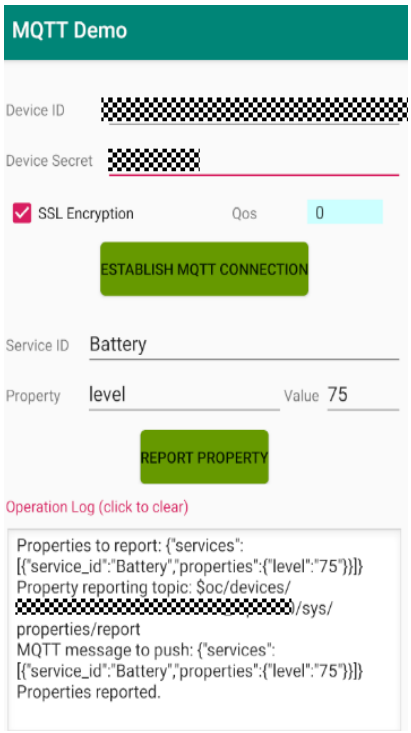
Receiving a Command

The **MainActivity** class provides the methods for receiving commands delivered by the platform. After an MQTT connection is established, you can deliver commands on the device details page of the **IoTDA console** or by using the **demo on the application side**. For example, deliver a command carrying the parameter name **command** and parameter value **5**. After the command is delivered, a result is received using the MQTT callback.

```
private final class MqttCallBack4IoTHub implements MqttCallbackExtended {
    ......
    @Overridepublic void messageArrived(String topic, MqttMessage message) throws Exception {
        Log.i(TAG, "Incoming message: " + new String(message.getPayload(), StandardCharsets.UTF_8));
        editText_mqtt_log.append("MQTT receives the delivered command: " + message + "\n")
    }
```

On the device details page, you can view the command delivery status. In this example, **timeout** is displayed because this demo does not return a response to the platform.

If the property reporting and command receiving are successful, the following information is displayed in the log area of the application:



3.8.5 C Demo Usage Guide

Overview

This topic uses C as an example to describe how to connect a device to the platform over MQTTS or MQTT and how to use **platform APIs** to report properties and subscribe to a topic for receiving commands.

NOTE

The code snippets in this document are only examples and are for trial use only. To put them into commercial use, obtain the IoT Device SDKs of the corresponding language for integration by referring to **Obtaining Resources**.

Prerequisites

- You have installed the Linux operating system (OS) and GCC (4.8 or later).
- You have obtained OpenSSL (required in MQTTS scenarios) and Paho library dependencies.
- You have obtained the device access address from the **IoTDA console**. For details, see **Platform Connection Information**.
- You have created a product and a device on the **IoTDA console**. For details, see **Creating a Product**, **Registering an Individual Device**, and **Registering a Batch of Devices**.

Preparations

- Compiling the OpenSSL library
  a. Visit the OpenSSL website (**https://www.openssl.org/source/**), download the latest OpenSSL version (for example, **openssl-1.1.1d.tar.gz**), upload it

to the Linux compiler (for example, to the **/home/test** directory), and run the following command to decompress the package:

```
tar -zxvf openssl-1.1.1d.tar.gz
```

b. Generate a **makefile**.

Run the following command to access the OpenSSL source code directory:

```
cd openssl-1.1.1d
```

Run the following configuration command:

```
./config shared --prefix=/home/test/openssl --openssldir=/home/test/openssl/ssl
```

In this command, **prefix** is the installation directory, **openssldir** is the configuration file directory, and **shared** is used to generate a dynamic-link library (**.so** library).

If an exception occurs during the compilation, add **no-asm** to the configuration command (indicating that the assembly code is not used).

./config no-asm shared --prefix=/home/test/openssl --openssldir=/home/test/openssl/ssl

```
[root@server-1908071538 test]# cd openssl-1.1.1d
[root@server-1908071538 openssl-1.1.1d]# ./config shared --prefix=/home/test/openssl --openssldir=/home/test/openssl/ssl
```

c. Generate library files.

Run the following command in the OpenSSL source code directory:

```
make depend
```

Run the following command for compilation:

```
make
```

Install OpenSSL.

```
make install
```

Find the **lib** directory in **home/test/openssl** under the OpenSSL installation directory.

The library files **libcrypto.so.1.1**, **libssl.so.1.1**, **libcrypto.so**, and **libssl.so** are generated. Copy these files to the **lib** folder of the demo and copy the content in **/home/test/openssl/include/openssl** to **include/openssl** of the demo.

Note: Some compilation tools are 32-bit. If these tools are used on a 64-bit Linux computer, delete **-m64** from the **makefile** before the compilation.

- Compiling the Eclipse Paho library file

a. Visit **https://github.com/eclipse/paho.mqtt.c** to download the source code **paho.mqtt.c**.

b. Decompress the package and upload it to the Linux compiler.

c. Modify the **makefile**.

i. Run the following command to edit the **makefile**:
```
vim Makefile
```

ii. Search for the string.
```
/DOXYGEN_COMMAND =
```

iii. Add the following two lines (customized OpenSSL header files and library files) under **/DOXYGEN_COMMAND =doxygen**:
```
CFLAGS += -I/home/test/openssl/include
LDFLAGS += -L/home/test/openssl/lib -lrt
```

```
127 INSTALL_PROGRAM = $(INSTALL)
128 INSTALL_DATA =  $(INSTALL) -m 644
129 DOXYGEN_COMMAND = doxygen
130 CFLAGS += -I/home/test/openssl/include
131 LDFLAGS += -L/home/test/openssl/lib -lrt
132
133 MAJOR_VERSION = 1
134 MINOR_VERSION = 0
135 VERSION = ${MAJOR_VERSION}.${MINOR_VERSION}
```

iv. Replace the OpenSSL addresses of **CCDLAGS_SO**, **LDFLAGS_CS**, **LDFLAGS_AS** and **FLAGS_EXES** to the actual ones.

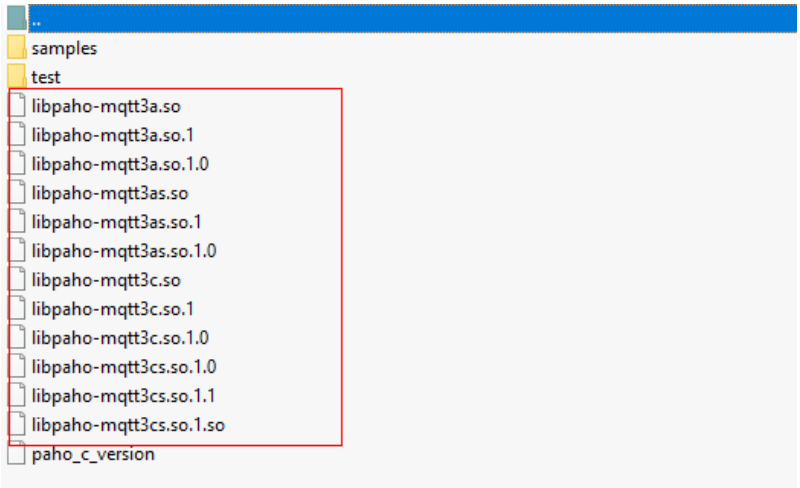
```
194
195 CCFLAGS_SO += -Wno-deprecated-declarations -DOSX -I /home/test/openssl/include
196 LDFLAGS_C += -Wl,-install_name,lib${MQTTLIB_C}.so.${MAJOR_VERSION}
197 LDFLAGS_CS += -Wl,-install_name,lib${MQTTLIB_CS}.so.${MAJOR_VERSION} -L /home/test/openssl/lib
198 LDFLAGS_A += -Wl,-install_name,lib${MQTTLIB_A}.so.${MAJOR_VERSION}
199 LDFLAGS_AS += -Wl,-install_name,lib${MQTTLIB_AS}.so.${MAJOR_VERSION} -L /home/test/openssl/lib
200 FLAGS_EXE += -DOSX
201 FLAGS_EXES += -L /home/test/openssl/lib
202
203 LDCONFIG = echo
204
```

d. Start the compilation.

i. Run the following command:
```
make clean
```

ii. Run the following command:
```
make
```

e. After the compilation is complete, you can view the libraries that are compiled in the **build/output** directory.



f. Copy the Paho library file.

Currently, only **libpaho-mqtt3as** is used in the SDK. Copy the **libpaho-mqtt3as.so** and **libpaho-mqtt3as.so.1** files to the **lib** folder of the demo. Go back to the Paho source code directory, and copy **MQTTAsync.h**, **MQTTClient.h**, **MQTTClientPersistence.h**, **MQTTProperties.h**, **MQTTReasonCodes.h**, and **MQTTSubscribeOpts.h** in the **src** directory to the **include/base** directory of the demo.

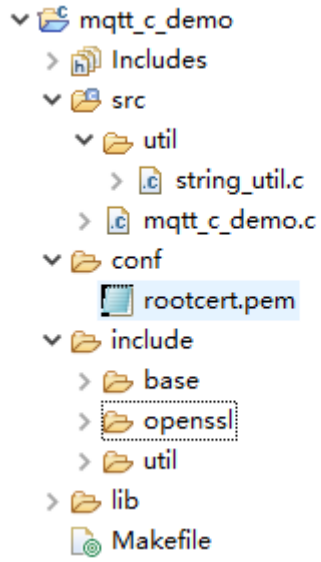
NOTE

Some Paho versions have the **MQTTExportDeclarations.h** header file. You are advised to add all MQTT-related header files to the folder.

Importing Sample Code

Step 1 Download the sample code **quickStart(C)**.

Step 2 Copy the code to the Linux runtime environment. The following figure shows the code file hierarchy.



Description of the directories:

- **src**: source code directory

  mqtt_c_demo: core source code of the demo

  util/string_util.c: utility resource file
- **conf**: certificate directory

  rootcert.pem is used by the device to verify the platform identity when the device connects to the platform. For not basic edition instance, copy the content of the **c/ap-southeast-1-device-client-rootcert.pem** file in the **certificate file** to the **conf/rootcert.pem** file.
- **include**: header files

  base: dependent Paho header files

  openssl: dependent OpenSSL header files

  util: header files of the dependent tool resources
- **lib**: dependent library file

  libcrypto.so*/libssl.so*: OpenSSL library file

  libpaho-mqtt3as.so*: Paho library file
- **Makefile**: Makefile

----End

Establishing a Connection

To connect a device or gateway to the platform, upload the device information to bind the device or gateway to the platform.

1. Set parameters.

```
char *uri = "ssl://iot-mqtts.cn-north-4.myhuaweicloud.com:8883";
int port = 8883;
char *username = "********"; //deviceId
char *password = "********";
```

   Note: MQTTS uses port 8883 for access. If MQTT is used for access, the URL is **tcp://***Domain name space*:**1883** and the port is 1883. For details about how to obtain the domain name space, see **Platform Connection Information**. The default heartbeat interval is 120 seconds. To change it, modify the **keepAliveInterval** parameter. For details about the heartbeat interval range, see **Constraints**.

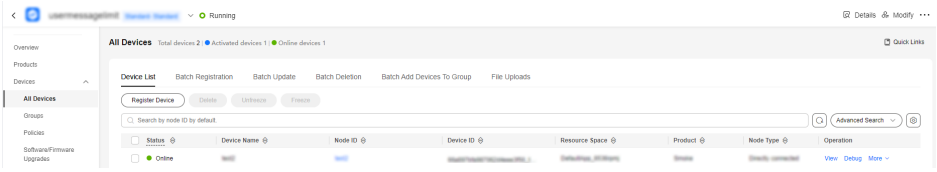
2. Start the connection.
   - Add **-lm** to the end of the 15th line in **Makefile** and run the **make** command for compilation. Delete **-m64** from the **makefile** in a 32-bit OS.
   - Run **export LD_LIBRARY_PATH=./lib/** to load the library file.
   - Run **./MQTT_Demo.o**.

```
//connect
int ret = mqtt_connect();
if (ret != 0) {
    printf("connect failed, result %d\n", ret);
}
```

3. If the connection is successful, the message "connect success" is displayed. The device is also displayed as **Online** on the console.

```
begin to connect the server.
connect success.
```

Figure 3-212 Device list - Device online status



If the connection fails, the mqtt_connect_failure function executes backoff reconnection. The example code is as follows:

```
void mqtt_connect_failure(void *context, MQTTAsync_failureData *response) {
    retryTimes++;
    printf("connect failed: messageId %d, code %d, message %s\n", response->token, response->code,
response->message);
    // Backoff reconnection
    int lowBound =  defaultBackoff * 0.8;
    int highBound = defaultBackoff * 1.2;
    int randomBackOff = rand() % (highBound - lowBound + 1);
    long backOffWithJitter = (int)(pow(2.0, (double)retryTimes) - 1) * (randomBackOff + lowBound);
    long waitTImeUntilNextRetry = (int)(minBackoff + backOffWithJitter) > maxBackoff ? (minBackoff
+ backOffWithJitter) : maxBackoff;

    TimeSleep(waitTImeUntilNextRetry);
```



```
    //connect
    int ret = mqtt_connect();
    if (ret != 0) {
        printf("connect failed, result %d\n", ret);
    }
}
```

Subscribing to a Topic

Only devices that subscribe to a specific topic can receive messages about the topic published by the broker. For details on the preset topics, see **Topics**.

Subscribe to a topic.

```
//subscribe
char *cmd_topic = combine_strings(3, "$oc/devices/", username, "/sys/commands/#");
ret = mqtt_subscribe(cmd_topic);
free(cmd_topic);
cmd_topic = NULL;
if (ret < 0) {
    printf("subscribe topic error, result %d\n", ret);
}
```

If the subscription is successful, the message "subscribe success" is displayed in the demo.

Reporting Properties

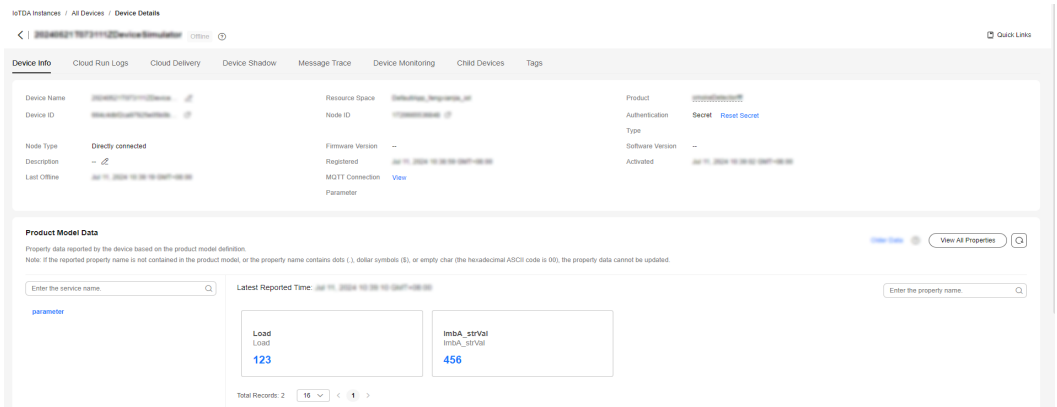
Devices can report their properties to the platform. For details, see **Reporting Device Properties**.

```
//publish data
char *payload = "{\"services\":[{\"service_id\":\"parameter\",\"properties\":{\"Load\":\"123\",\"ImbA_strVal
\":\"456\"}}]}";
char *report_topic = combine_strings(3, "$oc/devices/", username, "/sys/properties/report");
ret = mqtt_publish(report_topic, payload);
free(report_topic);
report_topic = NULL;
if (ret < 0) {
    printf("publish data error, result %d\n", ret);
}
```

If the property reporting is successful, the message "publish success" is displayed in the demo.

The reported properties are displayed on the device details page.

Figure 3-213 Viewing reported data - Parameter



NOTE

If the latest data is not displayed on the device details page, check whether the services and properties reported by the device are the same as those in the product model.

Receiving a Command

After subscribing to a command topic, you can deliver a synchronous command on the console. For details, see **Command Delivery to an Individual MQTT Device**.

If the command delivery is successful, the command received is displayed in the demo:

```
mqtt_message_arrive() success, the topic is $oc/devices/5ebac693352cfb02c567ec88_test2345/sys/commands/request_id=b5fb4352-43cb-43d7-9ab0-802c435e9ec8, the payload is {"paras":{"timeRead":"1"},"service_id":"command","command_name":"timeRead"}
```

The code for receiving commands in the demo is as follows:

```
//receive message from the server
int mqtt_message_arrive(void *context, char *topicName, int topicLen, MQTTAsync_message *message) {
    printf( "mqtt_message_arrive() success, the topic is %s, the payload is %s \n", topicName, message->payload);
    return 1; // cannot return 0 here, otherwise the message will not update or something wrong would happen
}
```

NOTE

Synchronous commands require device responses. For details, see **Upstream Response Parameters**.

3.8.6 C# Demo Usage Guide

Overview

This topic uses C# as an example to describe how to connect a device to the platform over MQTTS or MQTT and how to use **platform APIs** to report properties and subscribe to a topic for receiving commands.

NOTE

The code snippets in this document are only examples and are for trial use only. To put them into commercial use, obtain the IoT Device SDKs of the corresponding language for integration by referring to **Obtaining Resources**.

Prerequisites

- You have installed Microsoft Visual Studio. If not, follow the instructions provided in **Install Microsoft Visual Studio**.
- You have obtained the device access address from the **IoTDA console**. For details, see **Platform Connection Information**.
- You have created a product and a device on the **IoTDA console**. For details, see **Create a Product**, **Registering an Individual Device**, and **Registering a Batch of Devices**.

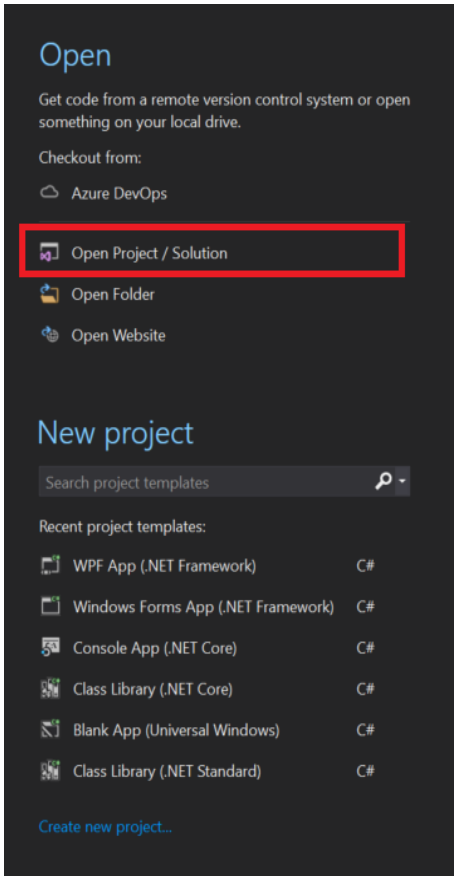
Preparations

- Go to the **Microsoft website** to download and install Microsoft Visual Studio of a desired version. (The following uses Windows 64-bit, Microsoft Visual Studio 2017, and .NET Framework 4.5.1 as examples.)
- After the download is complete, run the installation file and install Microsoft Visual Studio as prompted.

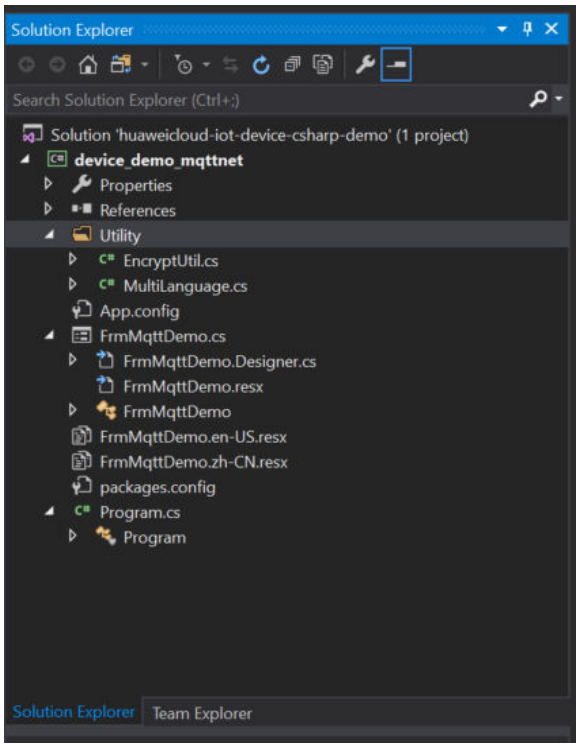
Importing Sample Code

Step 1 Download the sample code **quickStart(C#)**.

Step 2 Run Microsoft Visual Studio 2017, click **Open Project/Solution**, and select the sample code downloaded.



Step 3 Import the sample code.



Description of the directories:

- **App.config**: configuration file containing the server address and device information
- **C#**: C# code of the project

  EncryptUtil.cs: auxiliary class for device secret encryption

  FrmMqttDemo.cs: window UI

  Program.cs: entry for starting the demo
- **dll**: third-party libraries used in the project

  MQTTnet v3.0.11 is a high-performance, open-source .NET library based on MQTT. It supports both MQTT servers and clients. The reference library files include **MQTTnet.dll**.

  MQTTnet.Extensions.ManagedClient v3.0.11 is an extension library that uses MQTTnet to provide additional functions for the managed MQTT client.

Step 4 Set the project parameters in the demo.

- **App.config**: Set the server address, device ID, and device secret. When the demo is started, the information is automatically written to the demo main page.

  <add key="serverUri" value="serveruri"/>

  <add key="deviceId" value="deviceid"/>

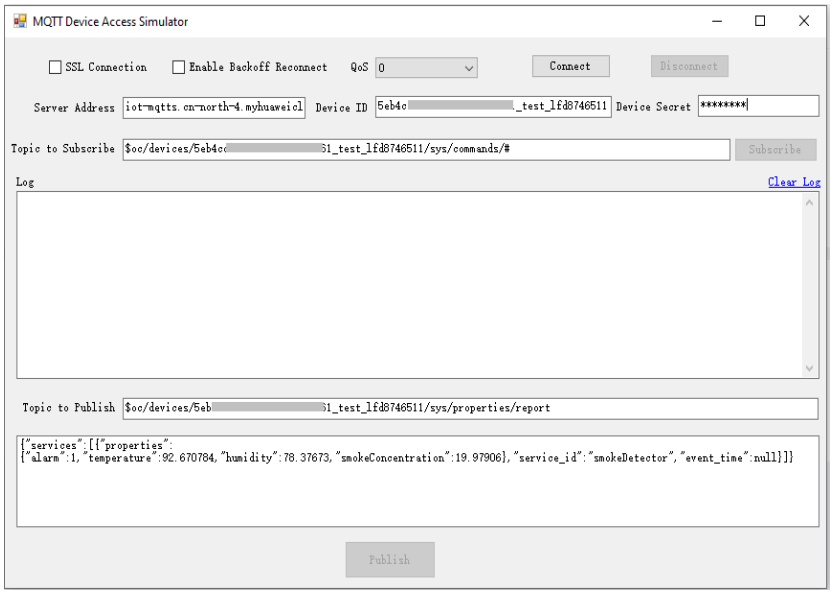
  <add key="deviceSecret" value="secret"/>

  <add key="PortIsSsl" value="8883"/>

  <add key="PortNotSsl" value="1883"/>

----End

UI Display



1. The **FrmMqttDemo** class provides a UI. By default, the **FrmMqttDemo** class automatically obtains the server address, device ID, and device secret from the **App.config** file after startup. Set the parameters based on the actual device information.
   - Server address: domain name. For details on how to obtain the domain name, see **Platform Connection Information**.
   - Device ID and secret: obtained after the **device is registered** on the IoTDA console or the API **Creating a Device** is called.
2. In the example, enter the server address. (The server address must match and be used together with the corresponding **certificate file** during SSL-encrypted access.)
   ```
   <add key="serverUri" value="iot-mqtts.cn-north-4.myhuaweicloud.com"/>;
   ```
3. Select SSL encryption or no encryption when establishing a connection on the device side and set the QoS mode to **0** or **1**. Currently, QoS 2 is not supported. For details, see **Constraints**.

Establishing a Connection

To connect a device or gateway to the platform, upload the device information to bind the device or gateway to the platform.

1. The **FrmMqttDemo** class provides methods for establishing MQTT or MQTTS connections. By default, MQTT uses port 1883, and MQTTS uses port 8883. (In the case of MQTTS connections, you must load the **DigiCertGlobalRootCA.crt.pem** certificate for verifying the platform identity. This certificate is used for login authentication when the device connects to the platform. You can download the certificate file from **Obtaining Resources**.) Call the **ManagedMqttClientOptionsBuilder** class to set the initial **KeepAlivePeriod**. The recommended heartbeat interval for MQTT connections is 120 seconds. For details, see **Constraints**.
   ```
   int portIsSsl = int.Parse(ConfigurationManager.AppSettings["PortIsSsl"]);
   int portNotSsl = int.Parse(ConfigurationManager.AppSettings["PortNotSsl"]);
   ```



```
if (client == null)
{
    client = new MqttFactory().CreateManagedMqttClient();
}

string timestamp = DateTime.Now.ToString("yyyyMMddHH");
string clientID = txtDeviceId.Text + "_0_0_" + timestamp;

// Encrypt passwords using HMAC SHA256.
string secret = string.Empty;
if (!string.IsNullOrEmpty(txtDeviceSecret.Text))
{
    secret = EncryptUtil.HmacSHA256(txtDeviceSecret.Text, timestamp);
}

// Check whether the connection is secure.
if (!cbSSLConnect.Checked)
{
    options = new ManagedMqttClientOptionsBuilder()
    .WithAutoReconnectDelay(TimeSpan.FromSeconds(RECONNECT_TIME))
    .WithClientOptions(new MqttClientOptionsBuilder()
        .WithTcpServer(txtServerUri.Text, portNotSsl)
        .WithCommunicationTimeout(TimeSpan.FromSeconds(DEFAULT_CONNECT_TIMEOUT))
        .WithCredentials(txtDeviceId.Text, secret)
        .WithClientId(clientID)
        .WithKeepAlivePeriod(TimeSpan.FromSeconds(DEFAULT_KEEPLIVE))
        .WithCleanSession(false)
        .WithProtocolVersion(MqttProtocolVersion.V311)
        .Build())
    .Build();
}
else
{
    string caCertPath = Environment.CurrentDirectory + @"\certificate\rootcert.pem";
    X509Certificate2 crt = new X509Certificate2(caCertPath);

    options = new ManagedMqttClientOptionsBuilder()
    .WithAutoReconnectDelay(TimeSpan.FromSeconds(RECONNECT_TIME))
    .WithClientOptions(new MqttClientOptionsBuilder()
        .WithTcpServer(txtServerUri.Text, portIsSsl)
        .WithCommunicationTimeout(TimeSpan.FromSeconds(DEFAULT_CONNECT_TIMEOUT))
        .WithCredentials(txtDeviceId.Text, secret)
        .WithClientId(clientID)
        .WithKeepAlivePeriod(TimeSpan.FromSeconds(DEFAULT_KEEPLIVE))
        .WithCleanSession(false)
        .WithTls(new MqttClientOptionsBuilderTlsParameters()
        {
            AllowUntrustedCertificates = true,
            UseTls = true,
            Certificates = new List<X509Certificate> { crt },
            CertificateValidationHandler = delegate { return true; },
            IgnoreCertificateChainErrors = false,
            IgnoreCertificateRevocationErrors = false
        })
        .WithProtocolVersion(MqttProtocolVersion.V311)
        .Build())
    .Build();
}
```

2. Call the **StartAsync** method in the **FrmMqttDemo** class to set up a connection. After the connection is set up, the **OnMqttClientConnected** is called to print connection success logs.

```
Invoke((new Action(() =>
{
    ShowLogs($"{"try to connect to server " + txtServerUri.Text}{Environment.NewLine}");
})));

if (client.IsStarted)
{
```



```
    await client.StopAsync();
}

// Register an event.
client.ApplicationMessageProcessedHandler = new
ApplicationMessageProcessedHandlerDelegate(new
Action<ApplicationMessageProcessedEventArgs>(ApplicationMessageProcessedHandlerMethod)); //
Called when a message is published.

client.ApplicationMessageReceivedHandler = new
MqttApplicationMessageReceivedHandlerDelegate(new
Action<MqttApplicationMessageReceivedEventArgs>(MqttApplicationMessageReceived)); // Called
when a command is delivered.

client.ConnectedHandler = new MqttClientConnectedHandlerDelegate(new
Action<MqttClientConnectedEventArgs>(OnMqttClientConnected)); // Called when a connection is set
up.

Callback function when the client.DisconnectedHandler = new
MqttClientDisconnectedHandlerDelegate(new
Action<MqttClientDisconnectedEventArgs>(OnMqttClientDisconnected)); // Called when a connection
is released.

// Connect to the platform.
await client.StartAsync(options);
```

If the connection fails, the OnMqttClientDisconnected function executes backoff reconnection. Sample code:

```
private void OnMqttClientDisconnected(MqttClientDisconnectedEventArgs e)
{
    try {
        Invoke((new Action(() =>
        {
            ShowLogs("mqtt server is disconnected" + Environment.NewLine);

            txtSubTopic.Enabled = true;
            btnConnect.Enabled = true;
            btnDisconnect.Enabled = false;
            btnPublish.Enabled = false;
            btnSubscribe.Enabled = false;
        })));

        if (cbReconnect.Checked)
        {
            Invoke((new Action(() =>
            {
                ShowLogs("reconnect is starting" + Environment.NewLine);
            })));

            // Backoff reconnection
            int lowBound = (int)(defaultBackoff * 0.8);
            int highBound = (int)(defaultBackoff * 1.2);
            long randomBackOff = random.Next(highBound - lowBound);
            long backOffWithJitter = (int)(Math.Pow(2.0, retryTimes)) * (randomBackOff + lowBound);
            long waitTImeUtilNextRetry = (int)(minBackoff + backOffWithJitter) > maxBackoff ?
maxBackoff : (minBackoff + backOffWithJitter);

            Invoke((new Action(() =>
            {
                ShowLogs("next retry time: " + waitTImeUtilNextRetry + Environment.NewLine);
            })));

            Thread.Sleep((int)waitTImeUtilNextRetry);

            retryTimes++;

            Task.Run(async () => { await ConnectMqttServerAsync(); });
        }
    }
```



```
    catch (Exception ex)
    {
        Invoke((new Action(() =>
        {
            ShowLogs("mqtt demo error: " + ex.Message + Environment.NewLine);
        })));
    }
}
```

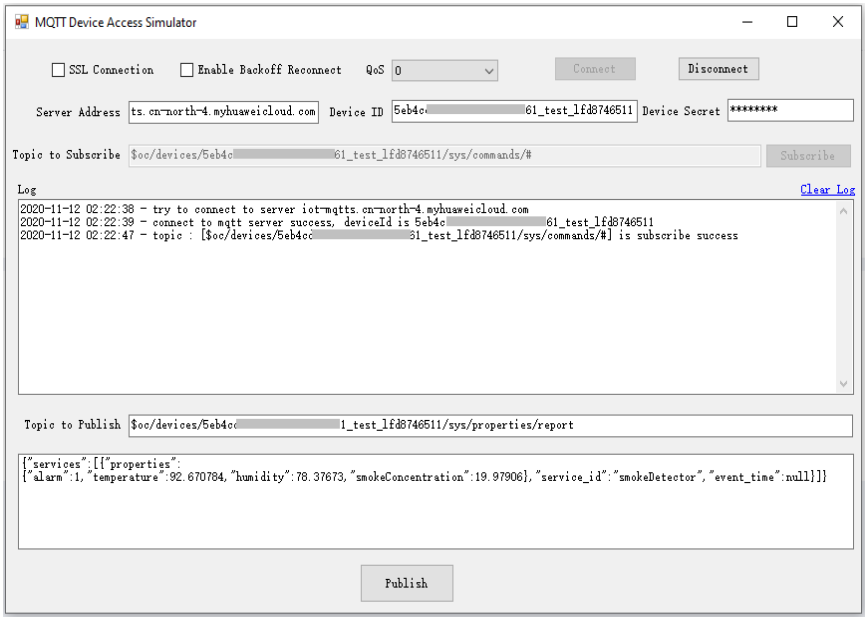
Subscribing to a Topic

Only devices that subscribe to a specific topic can receive messages about the topic published by the broker. For details on the preset topics, see **Topics**.

The **FrmMqttDemo** class provides the method for delivering subscription commands to topics.

```
List<MqttTopicFilter> listTopic = new List<MqttTopicFilter>();

var topicFilterBulderPreTopic = new MqttTopicFilterBuilder().WithTopic(topic).Build();
listTopic.Add(topicFilterBulderPreTopic);

// Subscribe to a topic.
client.SubscribeAsync(listTopic.ToArray()).Wait();
```

After the connection is established and a topic is subscribed, the following information is displayed in the log area on the home page of the demo:



Receiving a Command

The **FrmMqttDemo** class provides the method for receiving commands delivered by the platform. After an MQTT connection is established and a topic is subscribed, you can deliver a command on the device details page of the **IoTDA console** or by using the **demo on the application side**. After the command is delivered, the MQTT callback receives the command delivered by the platform.

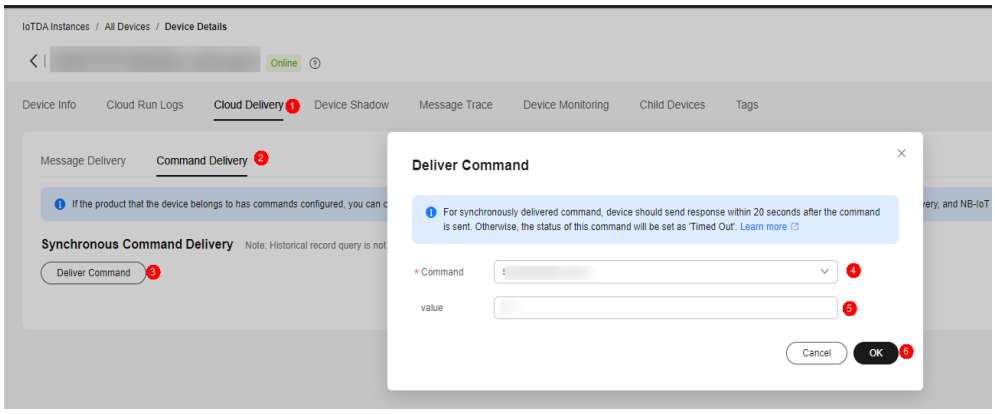
```
private void MqttApplicationMessageReceived(MqttApplicationMessageReceivedEventArgs e)
{
    Invoke((new Action(() =>
    {
```



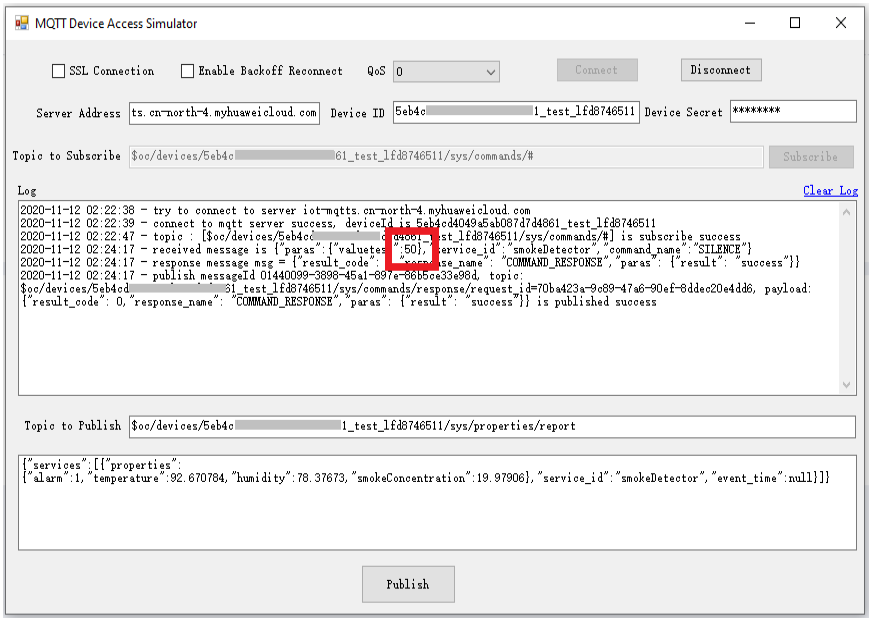
```
        ShowLogs($"received message is {Encoding.UTF8.GetString(e.ApplicationMessage.Payload)}
{Environment.NewLine}");

        string msg = "{\"result_code\": 0,\"response_name\": \"COMMAND_RESPONSE\",\"paras\": {\"result\":
\"success\"}}";

        string topic = "$oc/devices/" + txtDeviceId.Text + "/sys/commands/response/request_id=" +
e.ApplicationMessage.Topic.Split('=')[1];

        ShowLogs($"{"response message msg = " + msg}{Environment.NewLine}");

        var appMsg = new MqttApplicationMessage();
        appMsg.Payload = Encoding.UTF8.GetBytes(msg);
        appMsg.Topic = topic;
        appMsg.QualityOfServiceLevel = int.Parse(cbOosSelect.SelectedValue.ToString()) == 0 ?
MqttQualityOfServiceLevel.AtMostOnce : MqttQualityOfServiceLevel.AtLeastOnce;
        appMsg.Retain = false;

        // Return the upstream response.
        client.PublishAsync(appMsg).Wait();
    }));
}
```

For example, deliver a command carrying the parameter name **SmokeDetectorControl: SILENCE** and parameter value **50**.

Figure 3-214 Command delivery - Synchronous command delivery



After the command is delivered, the following information is displayed on the demo page:



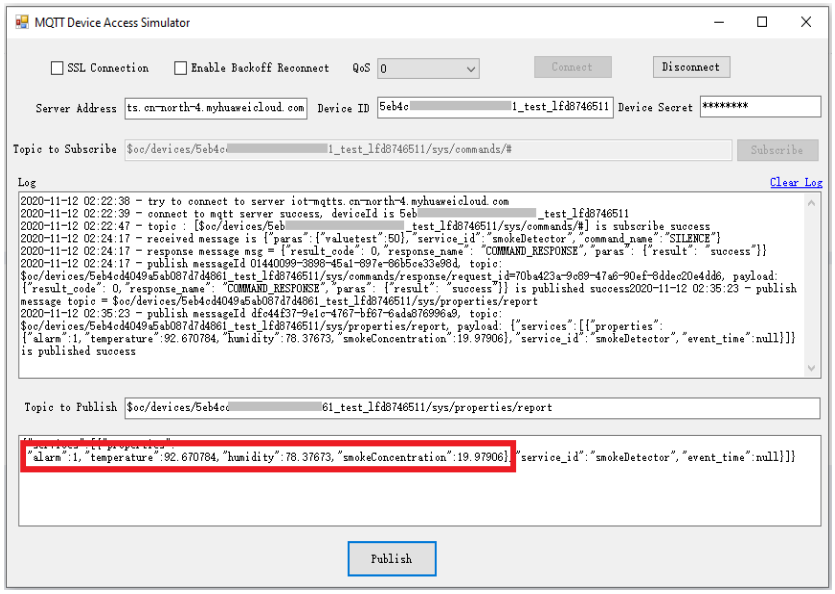
Publishing a Topic

Publishing a topic means that a device proactively reports its properties or messages to the platform. For details, see the API **Device Reporting Properties**.

The **FrmMqttDemo** class implements the property reporting topic and property reporting.

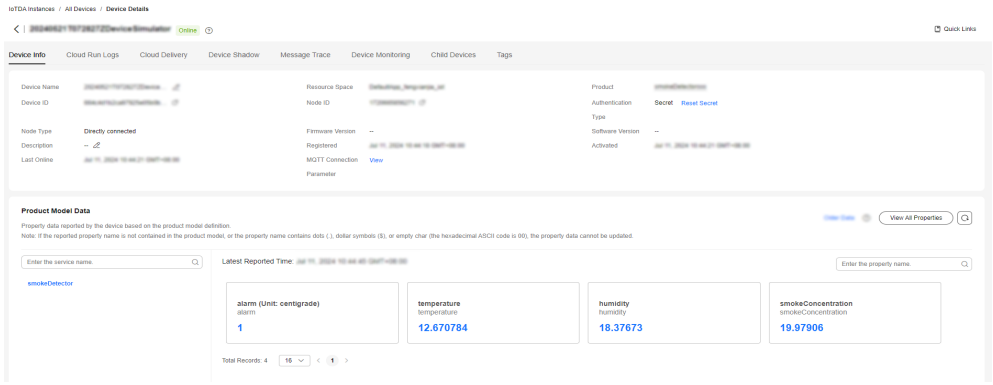
```
var appMsg = new MqttApplicationMessage();
appMsg.Payload = Encoding.UTF8.GetBytes(inputString);
appMsg.Topic = topic;
appMsg.QualityOfServiceLevel = int.Parse(cbOosSelect.SelectedValue.ToString()) == 0 ?
MqttQualityOfServiceLevel.AtMostOnce : MqttQualityOfServiceLevel.AtLeastOnce;
appMsg.Retain = false;

// Return the upstream response.
client.PublishAsync(appMsg).Wait();
```

After a topic is published, the following information is displayed on the demo page:



If the reporting is successful, the reported device properties are displayed on the device details page.

Figure 3-215 Viewing reported data - Demo_smokeDetector



NOTE

If the latest data is not displayed on the device details page, check whether the services and properties reported by the device are the same as those in the product model.

NOTE

Synchronous commands require device responses. For details, see **Upstream Response Parameters**.

3.8.7 Node.js Demo Usage Guide

Overview

This topic uses Node.js as an example to describe how to connect a device to the platform over MQTTS or MQTT and how to use **platform APIs** to report properties and subscribe to a topic for receiving commands.

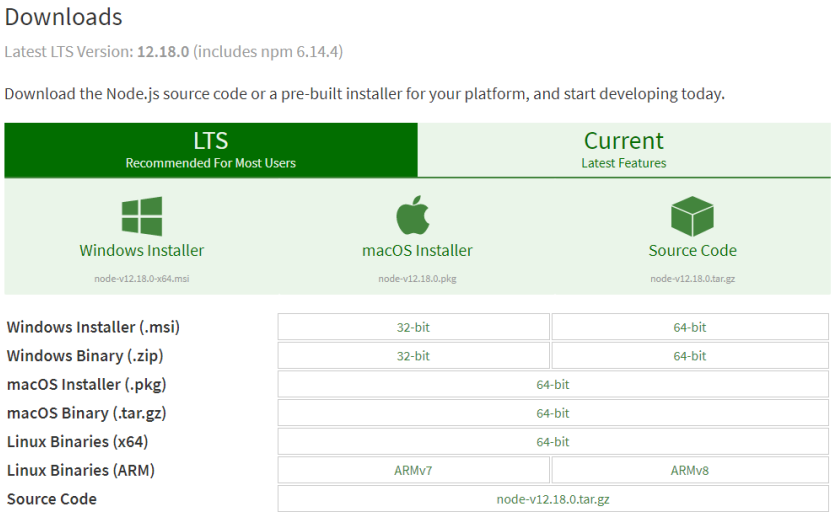
> **NOTE**
>
> The code snippets in this document are only examples and are for trial use only. To put them into commercial use, obtain the IoT Device SDKs of the corresponding language for integration by referring to **Obtaining Resources**.

Prerequisites

- You have installed Node.js by following the instructions provided in **Install Node.js**.
- You have obtained the device access address from the **IoTDA console**. For details, see **Platform Connection Information**.
- You have created a product and a device on the **IoTDA console**. For details, see **Creating a Product**, **Registering an Individual Device**, and **Registering a Batch of Devices**.

Preparations

1. Go to the **Node.js website** to download and install a desired version. The following uses Windows 64-bit and Node.js v12.18.0 (npm 6.14.4) as an example.



2. After the download is complete, run the installation file and install Node.js as prompted.
3. Verify that the installation is successful.

   Press **Win+R**, enter **cmd**, and press **Enter**. The command-line interface (CLI) is displayed.

   Enter **node -v** and press **Enter**. The Node.js version is displayed. Enter **npm -v**. If any version information is displayed, the installation is successful.

Importing Sample Code

Step 1 Download the sample code **quickStart(Node.js)** and decompress the package.

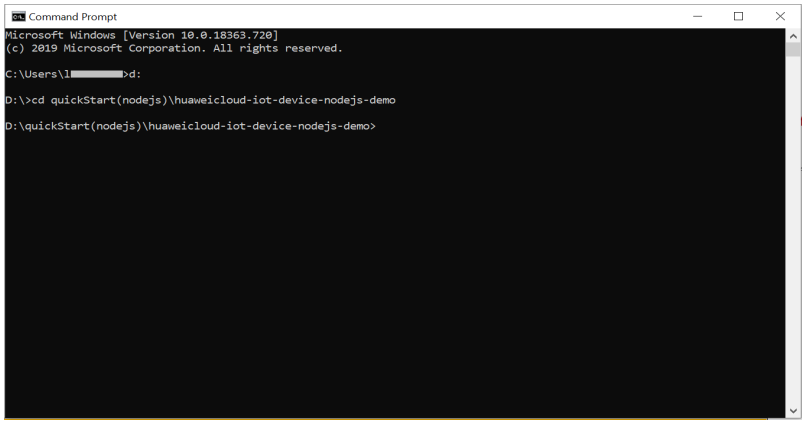
Step 2 Press **Win+R**, enter **cmd**, and press **Enter** to open the CLI. Run the following commands to install the global module:

npm install mqtt -g: This command is used to install the MQTT protocol module.

npm install crypto-js -g: This command is used to install the device secret cryptographic algorithm module.

npm install fs -g: This command is used to load the platform certificate.

Step 3 Find the directory where the package is decompressed.



Code directory:

- **DigiCertGlobalRootCA.crt.pem**: platform certificate file
- **MqttDemo.js**: Node.js source code for MQTT or MQTTS connection to the platform, property reporting, and command delivery.

Step 4 Set the project parameters in the demo. In **MqttDemo.js**, set the server address, device ID, and device secret for connecting to the device registered on the console when the demo is started.

- Server address: domain name. For details on how to obtain the server address, see **Platform Connection Information**. The server address must match and be used together with the corresponding **certificate file** during SSL-encrypted access.
- Device ID and secret: obtained after the **device is registered** on the IoTDA console or the API **Creating a Device** is called.

```
var TRUSTED_CA = fs.readFileSync("DigiCertGlobalRootCA.crt.pem");// Obtain a certificate.

// MQTT connection address of the platform (Replace it with the domain name of the IoT platform that the device is connected to.)
var serverUrl = "xxx.myhuaweicloud.com"; // Enter the access address of the platform that the device is connected to.

// Device ID and secret obtained during device registration (Replace them with the actual values.)
var deviceId = "722cb**********";
var secret = "****";
var timestamp = dateFormat("YYYYmmddHH", new Date());

var propertiesReportJson = {'services':[{'properties':
{'alarm':1,'temperature':12.670784,'humidity':18.37673,'smokeConcentration':19.97906},'service_id':'smokeDetector','event_time':null}]};
var responseReqJson = {'result_code': 0,'response_name': 'COMMAND_RESPONSE','paras': {'result': 'success'}};
```

Step 5 Select different options from **mqtt.connect(options)** to determine whether to perform SSL encryption during connection establishment on the device. You are advised to use the default MQTTS connection.

```
// MQTTS connection
var options = {
```



```
    host: serverUrl,
    port: 8883,
    clientId: getClientId(deviceId),
    username: deviceId,
    password:HmacSHA256(secret, timestamp).toString(),
    ca: TRUSTED_CA,
    protocol: 'mqtts',
    rejectUnauthorized: false,
    keepalive: 120,
    reconnectPeriod: 10000,
    connectTimeout: 30000
}

// MQTT connection is insecure and is not recommended.
var option = {
    host: serverUrl,
    port: 1883,
    clientId: getClientId(deviceId),
    username: deviceId,
    password: HmacSHA256(secret, timestamp).toString(),
    keepalive: 120,
    reconnectPeriod: 10000,
    connectTimeout: 30000
    //protocol: 'mqtts'
    //rejectUnauthorized: false
}

// By default, options is used for secure connection.
var client = mqtt.connect(options);
```

----End

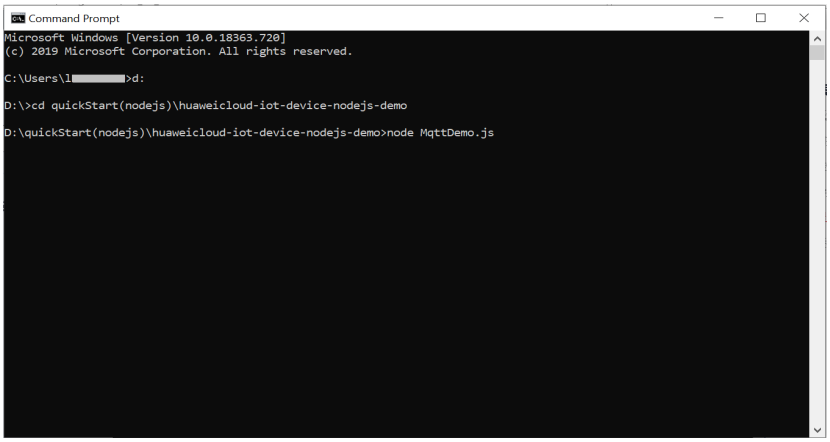
Starting the Demo

To connect a device or gateway to the platform, upload the device information to bind the device or gateway to the platform.

1. This demo provides methods such as establishing an MQTT or MQTTS connection. By default, MQTT uses port 1883, and MQTTS uses port 8883. (In the case of MQTTS connections, you must load the certificate for verifying the platform identity. The certificate is used for login authentication when the device connects to the platform.) Call the **mqtt.connect(options)** method to establish an MQTT connection.

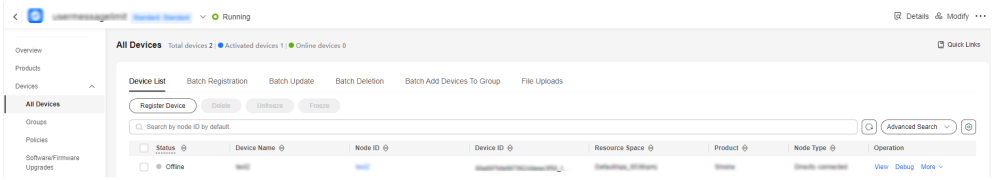
   ```
   var client = mqtt.connect(options);

   client.on('connect', function () {
       log("connect to mqtt server success, deviceId is " + deviceId);
       // Subscribe to a topic.
       subScribeTopic();
       // Publish a message.
       publishMessage();
   })

   // Respond to the command.
   client.on('message', function (topic, message) {
       log('received message is ' + message.toString());

       var jsonMsg = responseReq;
       client.publish(getResponseTopic(topic.toString().split("=")[1]), jsonMsg);
       log('response message is ' + jsonMsg);
   })
   ```

   Find the Node.js demo source code directory, modify **key project parameters**, and start the demo.



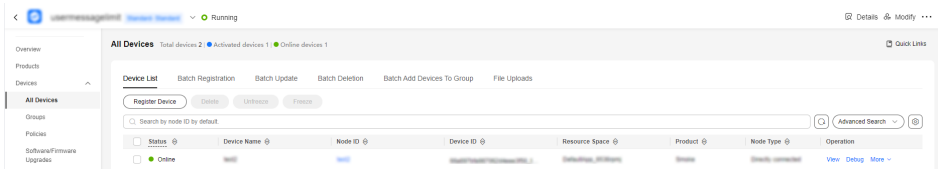
Before the demo is started, the device is in the offline state.

Figure 3-216 Device list - Device offline status



After the demo is started, the device status changes to online.

Figure 3-217 Device list - Device online status



If the connection fails, the reconnect function executes backoff reconnection. The example code is as follows:

```
client.on('reconnect', () => {

    log("reconnect is starting");

    // Backoff reconnection
    var lowBound = Number(defaultBackoff)*Number(0.8);
    var highBound = Number(defaultBackoff)*Number(1.2);

    var randomBackOff = parseInt(Math.random()*(highBound-lowBound+1),10);

    var backOffWithJitter = (Math.pow(2.0, retryTimes)) * (randomBackOff + lowBound);

    var waitTImeUtilNextRetry = (minBackoff + backOffWithJitter) > maxBackoff ? maxBackoff :
(minBackoff + backOffWithJitter);

    client.options.reconnectPeriod = waitTImeUtilNextRetry;

    log("next retry time: " + waitTImeUtilNextRetry);

    retryTimes++;
})
```


2. Only devices that subscribe to a specific topic can receive messages about the topic published by the broker. For details on the preset topics, see **Topics**. This demo calls the **subScribeTopic** method to subscribe to a topic. After the subscription is successful, wait for the platform to deliver a command.

```
// Subscribe to a topic for receiving commands.
function subScribeTopic() {
    client.subscribe(getCmdRequestTopic(), function (err) {
        if (err) {
            log("subscribe error:" + err);
        } else {
            log("topic : " + getCmdRequestTopic() + " is subscribed success");
        }
    })
}
```

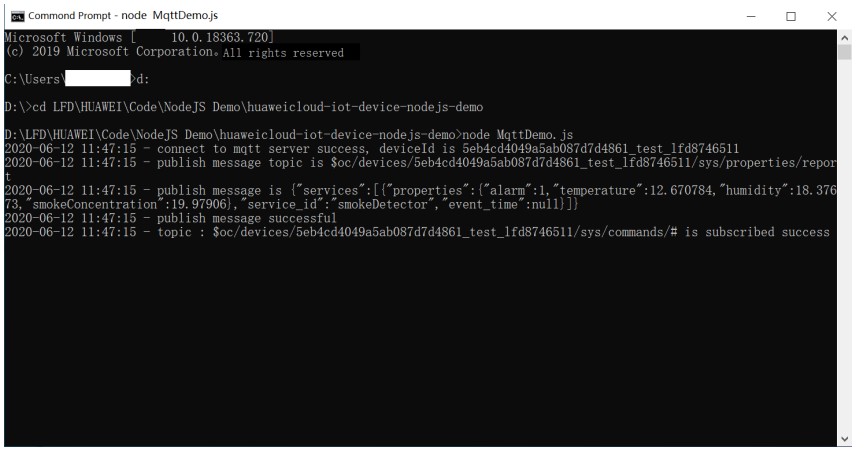
3. Publishing a topic means that a device proactively reports its properties or messages to the platform. For details, see the API **Device Reporting Properties**. After the connection is successful, call the **publishMessage** method to report properties.

```
// Report JSON data. serviceId must be the same as that defined in the product model.
function publishMessage() {
    var jsonMsg = propertiesReport;
    log("publish message topic is " + getReportTopic());
    log("publish message is " + jsonMsg);
    client.publish(getReportTopic(), jsonMsg);
    log("publish message successful");
}
```

Reported properties in the JSON format are as follows:

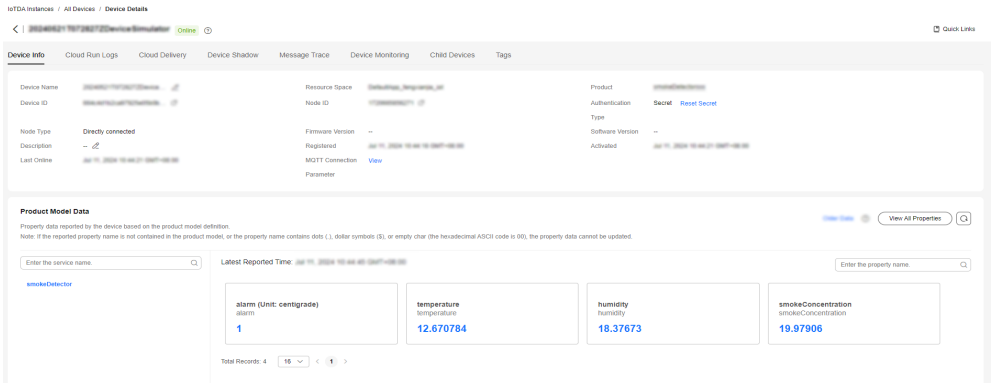
```
var propertiesReportJson = {'services':[{'properties':
{'alarm':1,'temperature':12.670784,'humidity':18.37673,'smokeConcentration':19.97906},'service_id':'smo
keDetector','event_time':null}]};
```

The following figure shows the CLI.



If the properties are reported, the following information is displayed on the IoTDA console:

Figure 3-218 Viewing reported data - Demo_smokeDetector



NOTE

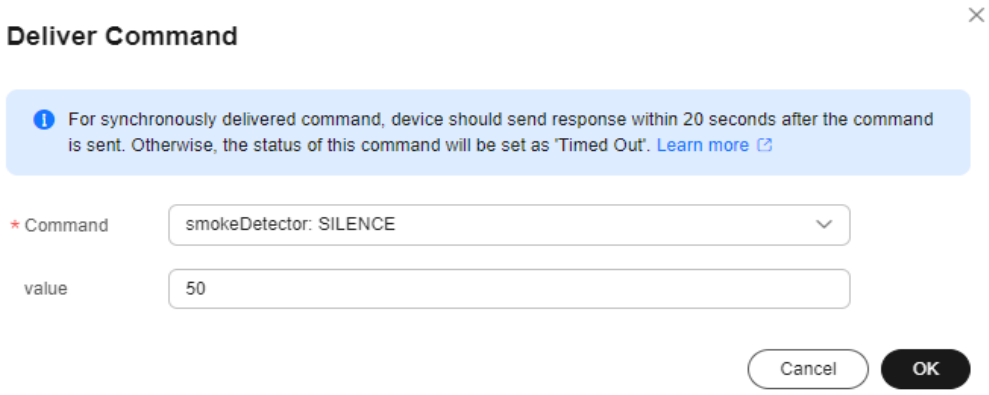
If the latest data is not displayed on the device details page, check whether the services and properties reported by the device are the same as those in the product model.

Receiving a Command

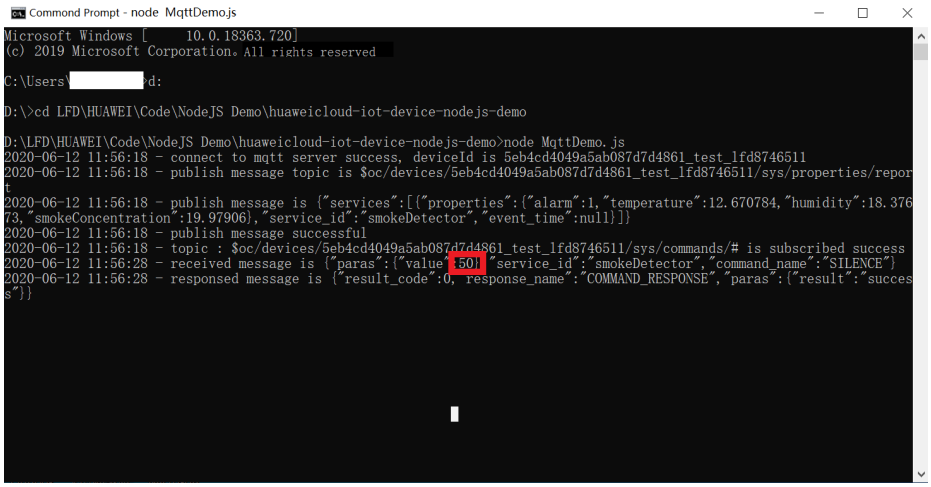
The demo provides the method for receiving commands delivered by the platform. After an MQTT connection is established and a topic is subscribed, you can deliver a command to a device of specific ID on the device details page of the **IoTDA console** or by using the **demo on the application side**. After the command is delivered, the MQTT callback function receives the command delivered by the platform.

For example, deliver a command carrying the parameter name **smokeDetector: SILENCE** and parameter value **50**.

Figure 3-219 Command delivery - SILENCE



After the command is delivered, the demo receives a 50 message. The following figure shows the command execution page.



NOTE

Synchronous commands require device responses. For details, see **Upstream Response Parameters**.

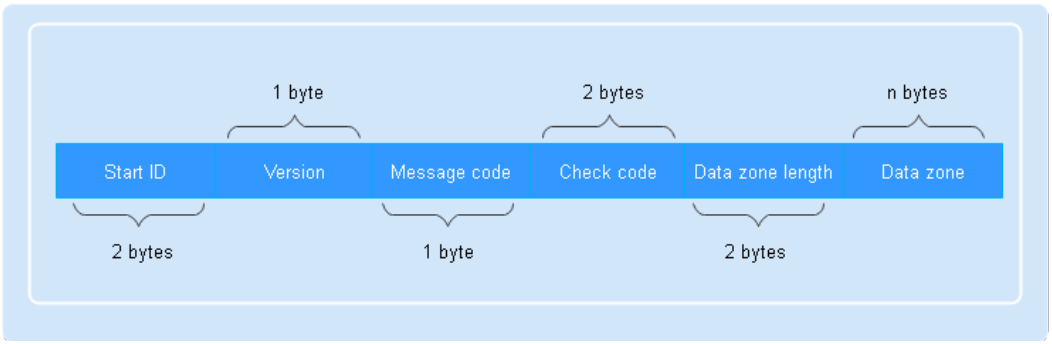
3.9 OTA Upgrade Adaptation on the Device Side

3.9.1 Adaptation Development on the Device Side

Overview

Software OTA is implemented using the Huawei proprietary **PCP protocol**. You must perform adaptation development on devices in accordance with the interaction process defined in the protocol. The following describes how a device constructs a PCP request and response based on the software upgrade interactions between the IoT platform and device. This helps you better develop software upgrade functions on the devices.

PCP requests and responses have the same message structure, as shown below.



For details on each field in the message structure, see the table below.

Field	Type	Description
Start ID	WORD	The value is fixed at **0XFFFE**.

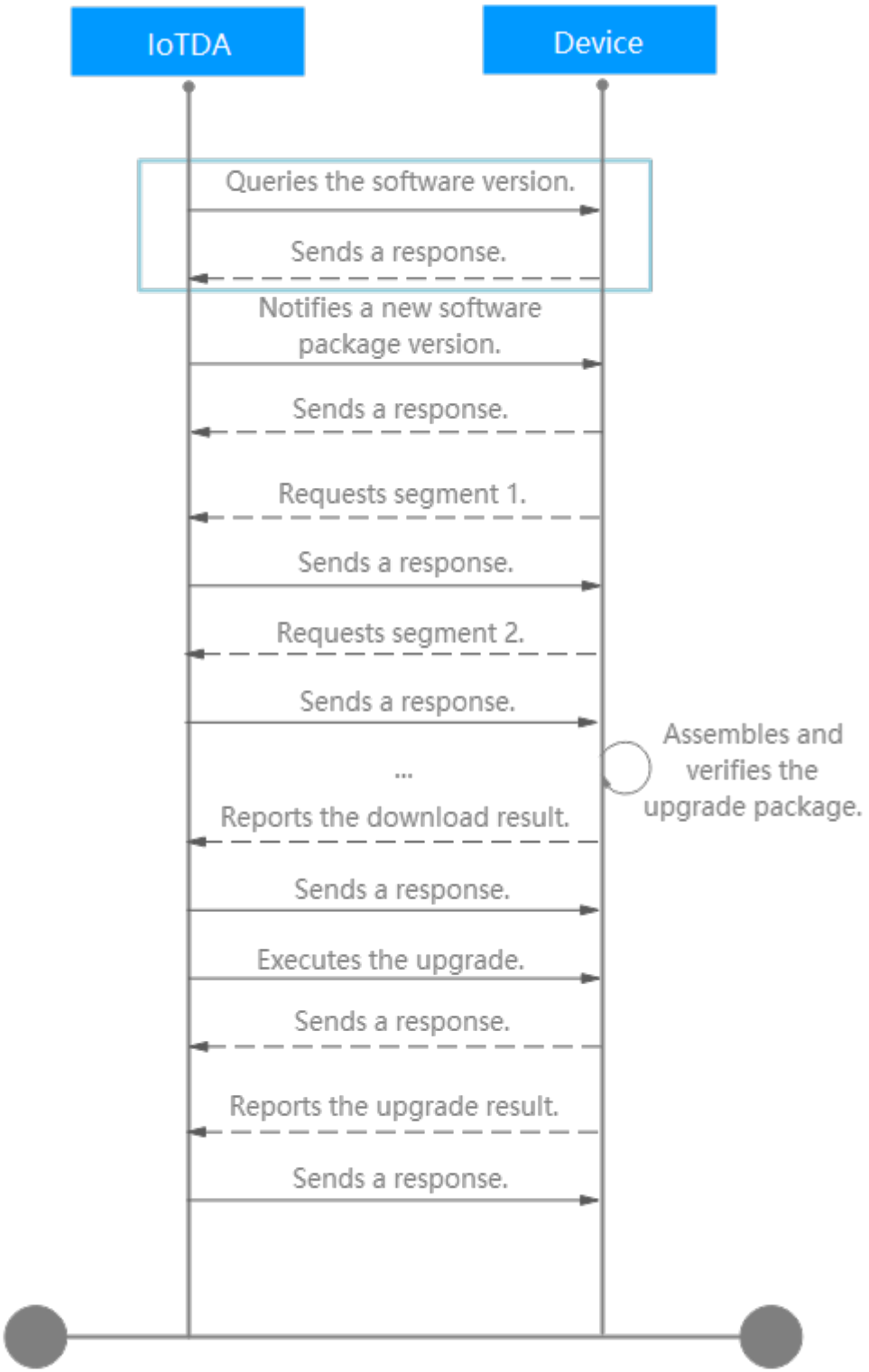
Field	Type	Description
Version	BYTE	The four most significant bits are reserved. The four least significant bits indicate the protocol version. Currently, the version is **1**.
Message code	BYTE	Type of the request exchanged between the platform and device. The message code of a response is the same as that of the request. The following message codes have been defined: • 0-18: reserved • 19: device version query • 20: software package notification • 21: software package download • 22: download result reporting • 23: upgrade execution • 24: upgrade result reporting • 25-127: reserved
Check code	WORD	CRC16 check value calculated from the start ID to the last byte of the data zone. Before the calculation, this field is set to **0**. The result is then written to the field after the CRC16 calculation. **NOTE** CRC16 algorithm: CRC16/CCITT x16+x12+x5+1
Data zone length	WORD	Length of the data zone.
Data zone	BYTE[n]	Variable length, which is defined by each instruction. For details, see the definitions of the request and response corresponding to each instruction.

Data Type	Description
BYTE	Unsigned 1-byte integer
WORD	Unsigned 2-byte integer
DWORD	Unsigned 4-byte integer
BYTE[n]	Hexadecimal number of *n* bytes

Data Type	Description
STRING	String

Query on the Device Version

In the software upgrade process, the platform delivers a version query request to the device and the device responds to the request. (The process below includes only the PCP interactions between the platform and device.)



Message Sent by the Platform

In accordance with the **PCP message structure**, the platform fills each field in the request as follows:

- **Start ID**: The value is fixed at the first two bytes of a message stream, that is, FFFE.

- **Version**: The value is a 1-byte integer and is fixed at 1 (hexadecimal value: 01).
- **Message code**: The value is a 1-byte integer. The message code for device version query is 19 (hexadecimal value: 13).
- **Check code**: The value is a 2-byte integer. The system sets the check code to 0000, calculates the complete message stream by using the CRC16 algorithm to obtain a new check code, and then replaces 0000 with the new code.
- **Data zone length**: The value is a 2-byte integer, indicating the length of the data zone. Based on the structure of the data zone, a version query request has no data zone. Therefore, the length is 0000.
- **Data zone**: indicates the data to be sent to the device. Based on the structure of the data zone, this message does not contain the data to send. The data zone field is null.

Field	Data Type	Description
No data zone		

Therefore, the combined code stream is FFFE 01 13 0000 0000. This stream is calculated using the CRC16 algorithm to obtain check code 4C9A. (The platform provides **CRC16 code examples** based on Java and C.) Then, the generated check code is used to replace 0000 in the original code stream to obtain FFFE 01 13 4C9A 0000. This code stream is sent by the platform to the device to query its version.

Message Sent by the Device

After receiving the version query request from the platform, the device returns the query result. The fields in the response are as follows:

- **Start ID**: The value is fixed at FFFE.
- **Version**: The value is fixed at 01.
- **Message code**: The value is 13 (the same as that in the request).
- **Check code**: The value 0000 is used before CRC16 calculation.
- **Data zone length**: In accordance with the data type of the fields in the data zone, the length is 17 bytes (hexadecimal value: 0011).
- **Data zone**: Based on the structure of the data zone, the result code of successful processing is 00. Assume that the version is V0.9, which is converted to ASCII characters 56302E39. The data type of the version is BYTE[16], which indicates 16 bytes. The version 56302E39 has only 4 bytes. Therefore, 0 is appended to obtain 56302E39000000000000000000000000. The data zone is 0056302E39000000000000000000000000.

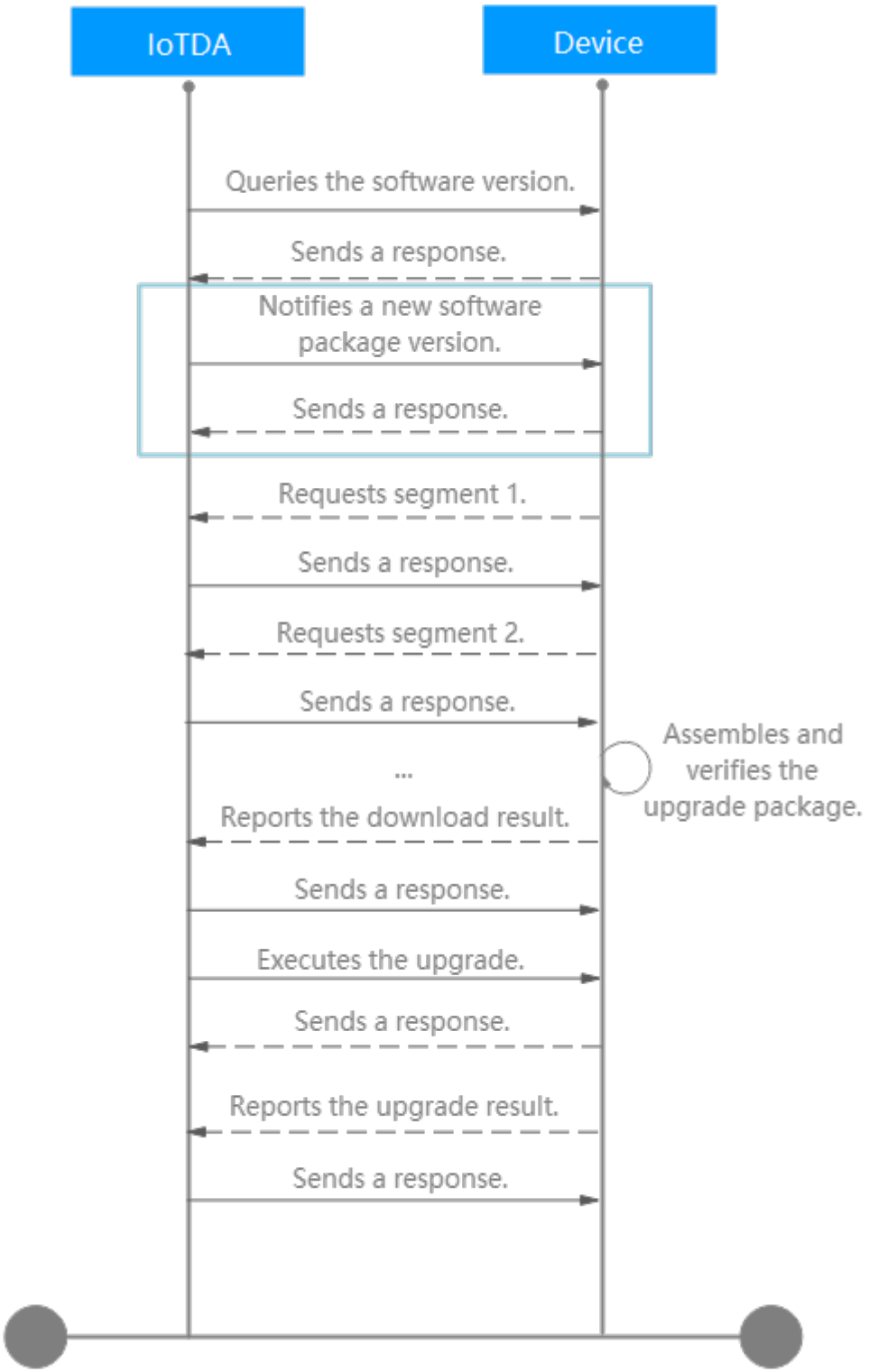
Field	Data Type	Description
Result code	BYTE	The value is **0X00**, indicating that the processing was successful.

Field	Data Type	Description
Current version	BYTE[16]	The version is described using ASCII characters. If there are not enough available digits, 0X00 is appended.

The combined code stream is FFFE 01 13 0000 0011 0056302E390000000000000000000000000. The check code after CRC16 calculation is 8DE3. Therefore, the device returns the code stream FFFE 01 13 8DE3 0011 0056302E390000000000000000000000000 to the platform.

Notification of a New Software Package

After obtaining the software version, the platform notifies the device of the software package of the new version.



Message Sent by the Platform

In accordance with the **PCP message structure**, the platform fills each field in the notification as follows:

- **Start ID**: The value is fixed at FFFE.
- **Version**: The value is fixed at 01.

- **Message code**: Based on the **message code**, the message code of the new software package notification is 20 (hexadecimal value: 14).
- **Check code**: The value 0000 is used before CRC16 calculation.
- **Data zone length**: In accordance with the data type of the fields in the data zone, the length is 22 bytes (hexadecimal value: 0016).
- **Data zone**:
  - **Target version**: The value consists of 16 bytes. If the target version is v1.0, the hexadecimal value appended with 0 is 56312E30000000000000000000000000.
  - **Upgrade package segment size**: The value consists of two bytes. You can manually enter the size of the upgrade package segment when uploading the software package. The default value is 500 bytes. The size ranges from 32 bytes to 500 bytes. For example, if the value is 500 bytes, the hexadecimal value is 01F4.
  - **Number of upgrade package segments**: The value consists of two bytes. The value is obtained by rounding up the result of the software package size divided by the segment size. If the software package size is 500 bytes, the number of segments is 1 (hexadecimal value: 0001).
  - **Check code**: The value consists of two bytes. This field has been deprecated. The fixed value is 0000.

Field	Data Type	Description
Target version	BYTE[16]	The version is described using ASCII characters. If there are not enough available digits, 0X00 is appended.
Upgrade package segment size	WORD	Size of each segment.
Number of upgrade package segments	WORD	Number of upgrade package segments.
Check code	WORD	The value is fixed at **0000**.

The combined code stream is FFFE 01 14 0000 0016 56312E30000000000000000000000000 01F4 0001 0000. The check code after CRC16 calculation is 02F7. Therefore, the code stream in the message sent by the platform to instruct the device to download the new software package is FFFE 01 14 02F7 0016 56312E30000000000000000000000000001F400010000.

Message Sent by the Device

After receiving the notification, the device returns a response to the platform, indicating whether to allow the upgrade. The fields in the response are as follows:

- **Start ID**: The value is fixed at FFFE.
- **Version**: The value is fixed at 01.

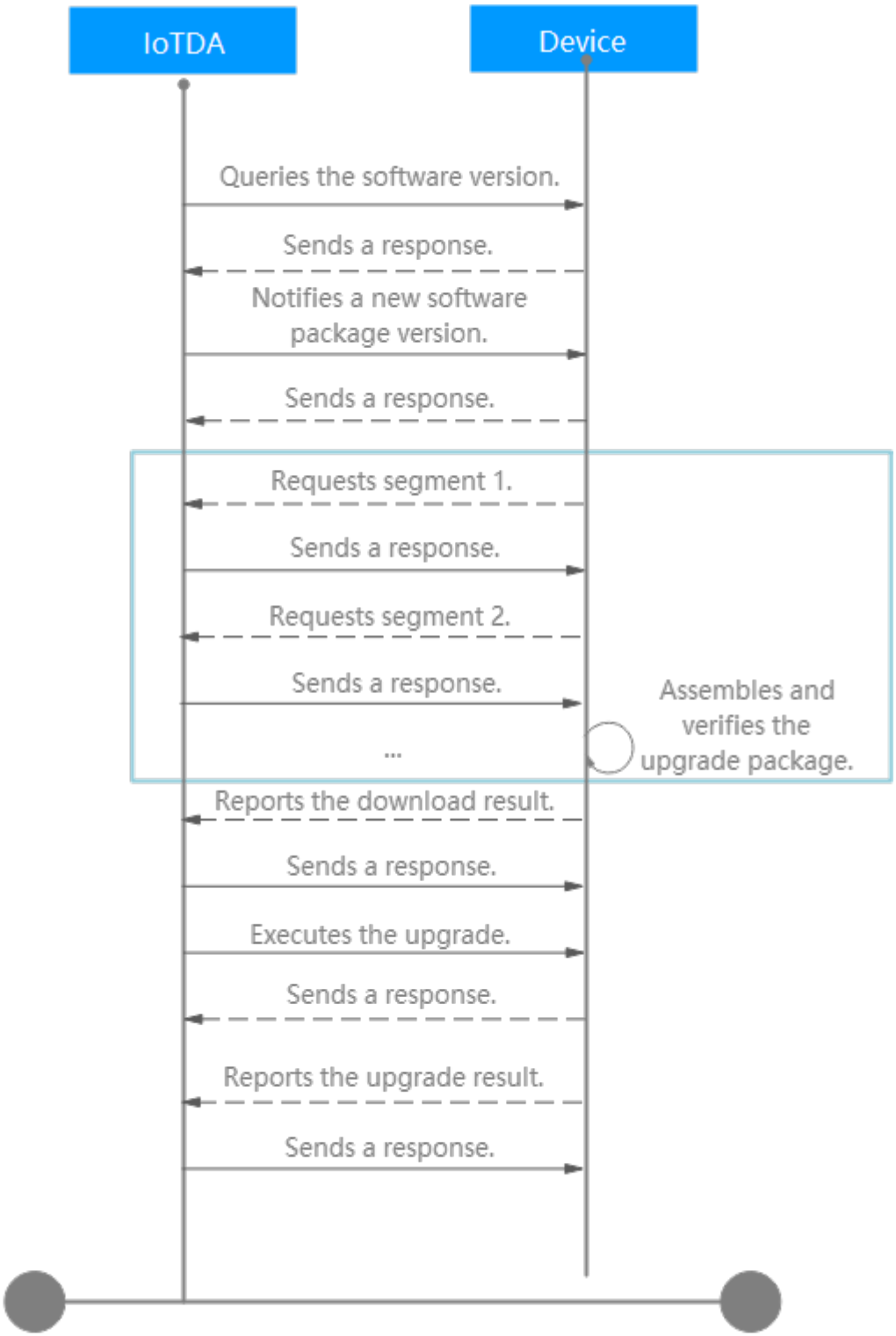
- **Message code**: The value is 14 (the same as that in the request).
- **Check code**: The value 0000 is used before CRC16 calculation.
- **Data zone length**: In accordance with the data type of the fields in the data zone, the length is 1 byte (hexadecimal value: 0001).
- **Data zone**: The device responds to the new software package notification based on the actual situation. In this example, the device responds with "The upgrade is allowed". The data zone is 00. The other result codes must be adapted accordingly.

Field	Data Type	Description
Result code	BYTE	**0X00**: The upgrade is allowed. **0X01**: The device is in use. **0X02**: The signal is weak. **0X03**: The latest version is in use. **0X04**: The battery power is low. **0X05**: The remaining space is insufficient. **0X09**: The memory is insufficient. **0X7F**: An internal error has occurred.

The combined code stream is FFFE 01 14 0000 0001 00. The check code after CRC16 calculation is D768. Therefore, the code stream in the message returned by the device is FFFE 01 14 D768 000100.

Downloading the Software Package

After the platform notifies the device of the new software package, the device requests to download the package according to the sequence number of each segment.



Message Sent by the Device

The device sends the first message to the platform to request packet segmentation. In accordance with the **PCP message structure**, the device fills each field in the first message as follows:

- **Start ID**: The value is fixed at FFFE.

- **Version**: The value is fixed at 01.
- **Message code**: In accordance with the **message code**, the message code for requesting the software package is 21 (hexadecimal value: 15).
- **Check code**: The value 0000 is used before CRC16 calculation.
- **Data zone length**: In accordance with the data type of the fields in the data zone, the length is 18 bytes (hexadecimal value: 0012).
- **Data zone**: The target version is the version in the notification delivered by the platform, v1.0 (hexadecimal value: 56312E30000000000000000000000000). The segment sequence number is 0 (hexadecimal value: 0000).

Field	Data Type	Description
Target version	BYTE[16]	The version is described using ASCII characters. If there are not enough available digits, 0X00 is appended.
Segment sequence number	WORD	Sequence number of the requested segment. The value starts from 0. The total number of segments is obtained by rounding up the result of the software package size divided by the segment size. The device can save the received segments and request for the missing segments next time. Resumable download is supported.

The combined code stream is FFFE 01 15 0000 0012 56312E30000000000000000000000000 0000. The check code after CRC16 calculation is 5618. Therefore, the code stream in the first segment request sent by the device is FFFE 01 15 5618 0012 56312E30000000000000000000000000.

For the code stream in other segment requests, only the segment sequence number needs to be replaced, and the check code needs to be replaced after CRC16 calculation. Details are not provided.

Message Sent by the Platform

After receiving a segment request, the platform delivers the segmented data to the device. The fields in the response to the first segment request are as follows:

- **Start ID**: The value is fixed at FFFE.
- **Version**: The value is fixed at 01.
- **Message code**: The value is 15 (the same as that in the request).

- **Check code**: The value 0000 is used before CRC16 calculation.
- **Data zone**: The result code is **00**. The segment sequence number is 0000. The segment data depends on the content defined in the software package. If the software package content is **HELLO, IoT SOTA!**, the hexadecimal value converted through ASCII code is 48454C4C4F2C20496F5420534F544121, 16 bytes in total. When uploading a software package, you need to manually enter the size of the upgrade package segment, which is 500 bytes. In this case, no 0 needs to be appended.
- **Data zone length**: In accordance with the data type of the fields in the data zone, the length is 19 bytes (hexadecimal value: 0013).

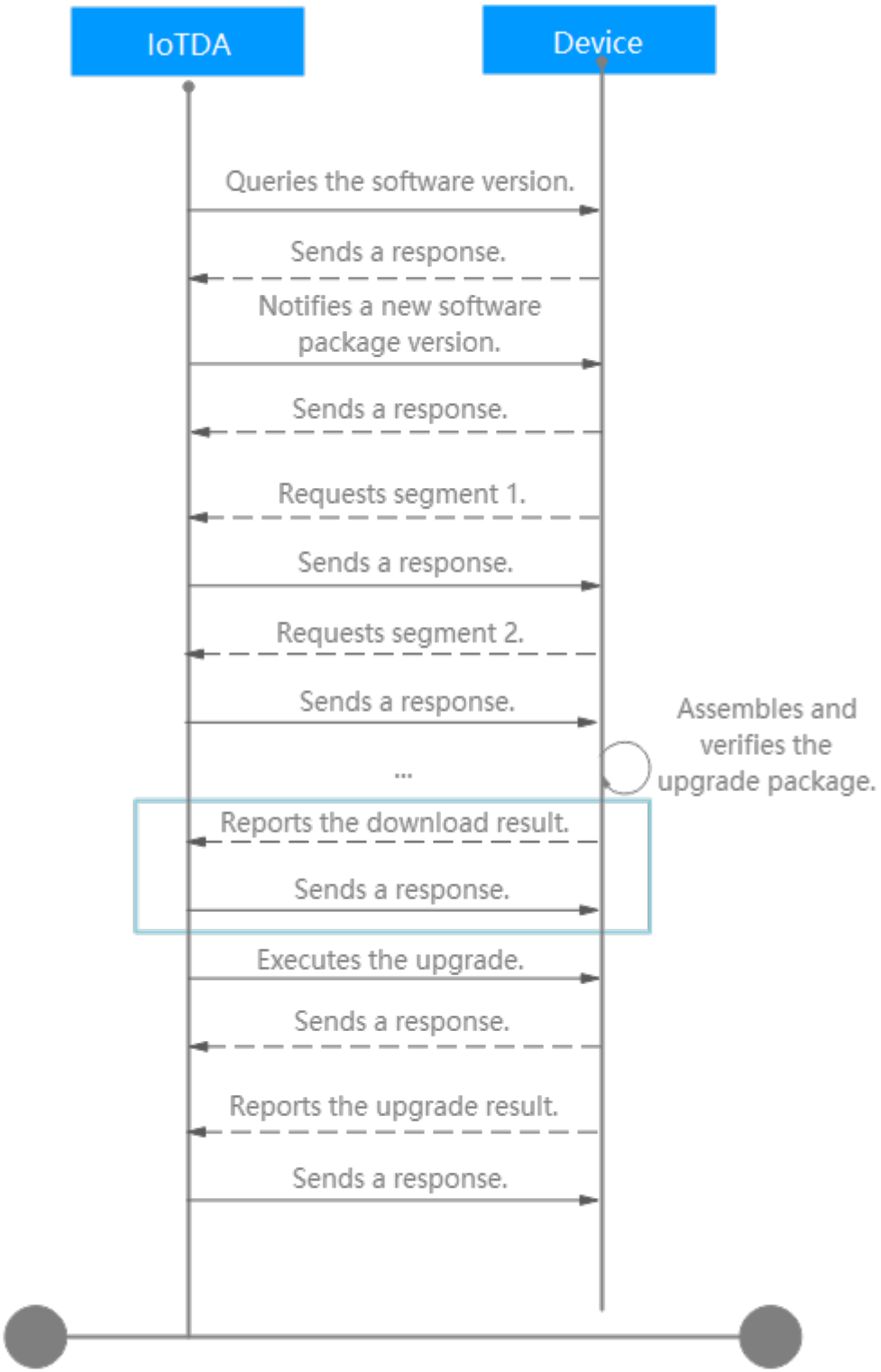
Field	Data Type	Description
Result code	BYTE	**0X00**: The processing was successful. **0X80**: The upgrade task does not exist. **0X81**: The specified segment does not exist.
Segment sequence number	WORD	Sequence number of a returned segment.
Segment data	BYTE[n]	Content of the segment. **n** indicates the segment size. If the result code is not 0, this field is not included.

The combined code stream is FFFE 01 15 0000 0013 00 0000 48454C4C4F2C20496F5420534F544121. The check code after CRC16 calculation is E107. The code stream in the message sent by the platform to respond to the first segment request is FFFE 01 15 E107 0013 00 0000 48454C4C4F2C20496F5420534F544121.

For the code stream in responses to the other segment requests, the segment sequence number and segment data need to be replaced, and the check code needs to be replaced after CRC16 calculation. Details are not provided.

Download Result Reporting

After receiving all segments and assembling them, the device reports the download result to the platform.



Message Sent by the Device

In accordance with the **PCP message structure**, the device fills each field in the message as follows:

- **Start ID**: The value is fixed at FFFE.
- **Version**: The value is fixed at 01.

- **Message code**: The value is 16 (the same as that in the request).
- **Check code**: The value 0000 is used before CRC16 calculation.
- **Data zone length**: In accordance with the data type of the fields in the data zone, the length is 1 byte (hexadecimal value: 0001).
- **Data zone**: carries the software package download results. For example, if the download was successful, the device reports 00.

Field	Data Type	Description
Download status	BYTE	**0X00**: The upgrade package has been downloaded. **0X05**: The remaining space is insufficient. **0X06**: The download timed out. **0X07**: The upgrade package failed to be verified. **0X08**: The upgrade package is not supported.

The combined code stream is FFFE 01 16 0000 0001 00. The check code after CRC16 calculation is 850E. The code stream in the download result message sent by the device is FFFE 01 16 850E 0001 00.

Message Sent by the Platform

After receiving the software package download results from the device, the platform returns a response. The fields in the response are as follows:

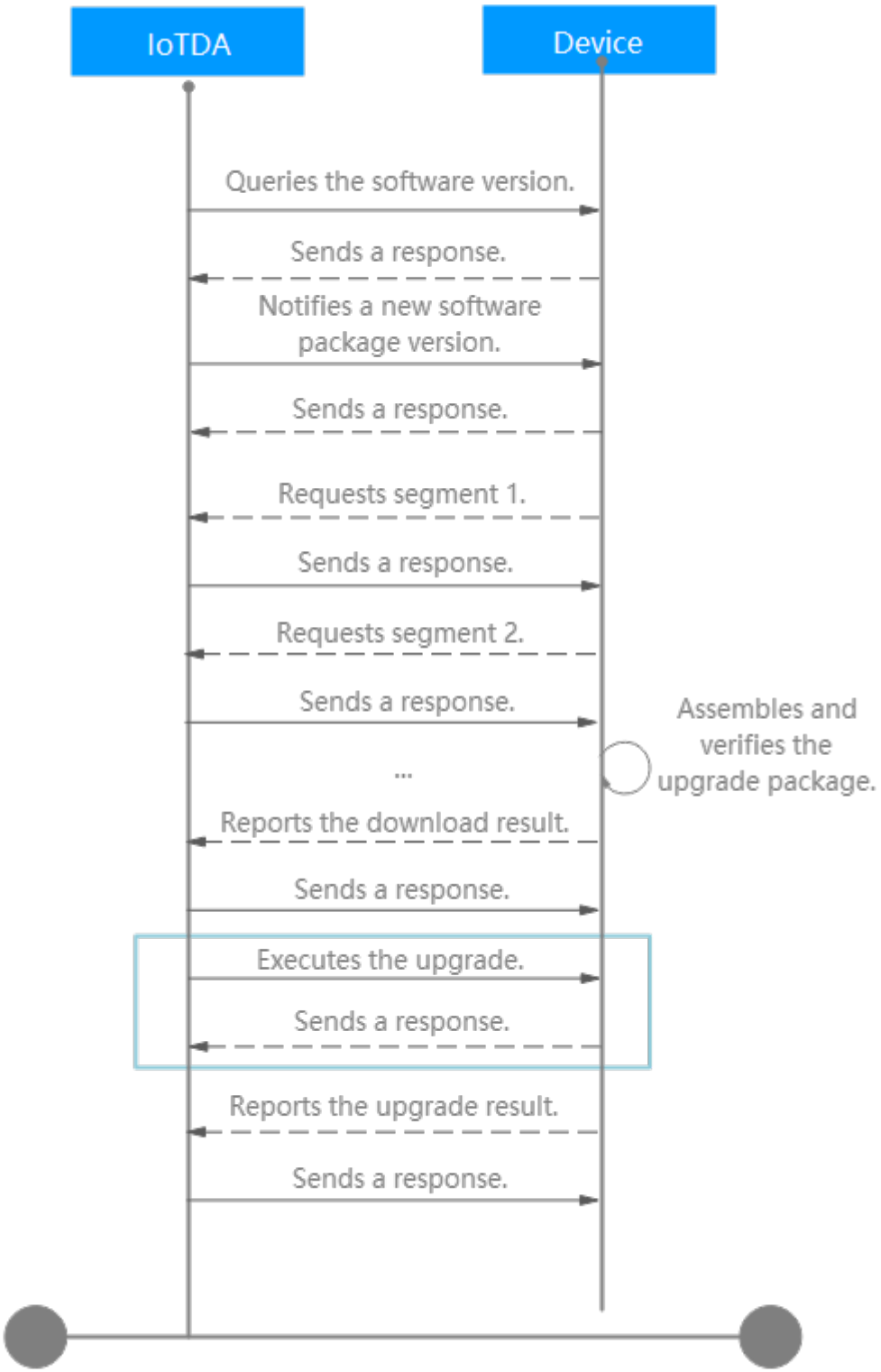
- **Start ID**: The value is fixed at FFFE.
- **Version**: The value is fixed at 01.
- **Message code**: The value is 16 (the same as that in the request).
- **Check code**: The value 0000 is used before CRC16 calculation.
- **Data zone length**: In accordance with the data type of the fields in the data zone, the length is 1 byte (hexadecimal value: 0001).
- **Data zone**: If the processing is successful, 00 is returned. If the processing fails, 80 is returned. In this example, 00 is returned.

Field	Data Type	Description
Result code	BYTE	**0X00**: The processing was successful. **0X80**: The upgrade task does not exist.

The combined code stream is FFFE 01 16 0000 0001 00. The check code after CRC16 calculation is 850E. The code stream in the message sent by the platform is FFFE 01 16 850E 0001 00.

Upgrade Execution

After receiving the software package download result from the device, the platform instructs the device to start the upgrade.



Message Sent by the Platform

In accordance with the **PCP message structure**, the platform fills each field in the instruction as follows:

- **Start ID**: The value is fixed at FFFE.
- **Version**: The value is fixed at 01.

- **Message code**: The value is 17 (the same as that in the request).
- **Check code**: The value 0000 is used before CRC16 calculation.
- **Data zone length**: In accordance with the data type of the fields in the data zone, the length is 0 bytes (hexadecimal value: 0000).
- **Data zone**: This field is not carried.

Field	Data Type	Description
No data zone		

The combined code stream is FFFE 01 17 0000 0000. The check code after CRC16 calculation is CF90. The code stream in the message sent by the platform is FFFE 01 17 CF90 0000.

Message Sent by the Device

After receiving the upgrade execution message from the platform, the device responds to the message. The fields in the message are as follows:

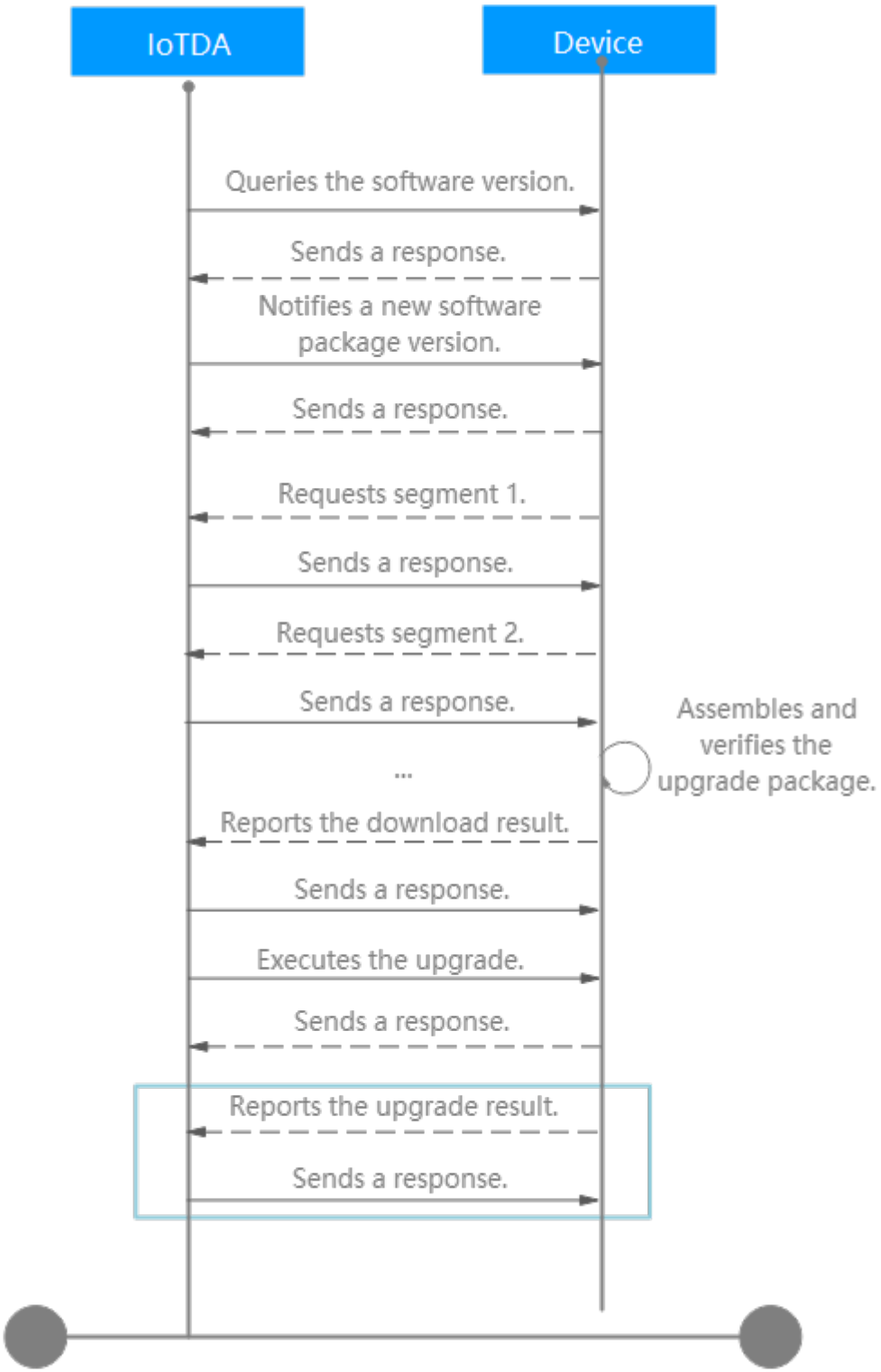
- **Start ID**: The value is fixed at FFFE.
- **Version**: The value is fixed at 01.
- **Message code**: The value is 17 (the same as that in the request).
- **Check code**: The value 0000 is used before CRC16 calculation.
- **Data zone length**: In accordance with the data type of the fields in the data zone, the length is 1 byte (hexadecimal value: 0001).
- **Data zone**: If the processing is successful, 00 is returned. For other processing results, see the data zone definition. In this example, 00 is returned.

Field	Data Type	Description
Result code	BYTE	**0X00**: The processing was successful. **0X01**: The device is in use. **0X04**: The battery power is low. **0X05**: The remaining space is insufficient. **0X09**: The memory is insufficient.

The combined code stream is FFFE 01 17 0000 0001 00. The check code after CRC16 calculation is B725. The code stream in the message returned by the device is FFFE 01 17 B725 0001 00.

Reporting the Upgrade Result

After executing the software upgrade, the device reports the upgrade result to the platform.



Message Sent by the Device

In accordance with the **PCP message structure**, the platform fills each field in an upgrade result message as follows:

- **Start ID**: The value is fixed at FFFE.
- **Version**: The value is fixed at 01.

- **Message code**: The value is 18 (the same as that in the request).
- **Check code**: The value 0000 is used before CRC16 calculation.
- **Data zone length**: In accordance with the data type of the fields in the data zone, the length is 17 bytes (hexadecimal value: 0011).
- **Data zone**: carries the result code and current version. In this example, the result code is 00, indicating that the upgrade was successful. The current version is the same as the software version delivered by the platform, v1.0 (hexadecimal value: 56312E30000000000000000000000000).

Field	Data Type	Description
Result code	BYTE	**0X00**: The upgrade was successful. **0X01**: The device is in use. **0X04**: The battery power is low. **0X05**: The remaining space is insufficient. **0X09**: The memory is insufficient. **0X0A**: The upgrade package failed to be installed. **0X7F**: An internal error has occurred.
Current version	BYTE[16]	Current version of the device.

The combined code stream is FFFE 01 18 0000 0011 0056312E30000000000000000000000000. The check code after CRC16 calculation is C7D2. The code stream in the upgrade result message reported by the device is FFFE 01 18 C7D2 0011 0056312E30000000000000000000000000.

Message Sent by the Platform

After receiving the upgrade result message, the platform responds to the device. The fields of each message are as follows:

- **Start ID**: The value is fixed at FFFE.
- **Version**: The value is fixed at 01.
- **Message code**: The value is 18 (the same as that in the request).
- **Check code**: The value 0000 is used before CRC16 calculation.
- **Data zone length**: In accordance with the data type of the fields in the data zone, the length is 1 byte (hexadecimal value: 0001).
- **Data zone**: If the processing is successful, 00 is returned. If the upgrade task does not exist, 80 is returned. In this example, 00 is returned.

Field	Data Type	Description
Result code	BYTE	**0X00**: The processing was successful. **0X80**: The upgrade task does not exist.

The combined code stream is FFFE 01 18 0000 0001 00. The check code after CRC16 calculation is AFA1. The code stream in the response returned by the platform is FFFE 01 18 AFA1 0001 00.

CRC16 Code Examples

Code example using the Java-based CRC16 algorithm:

```
public class CRC16 {

    /*
     * CCITT standard CRC16(1021) remainder table CRC16-CCITT ISO HDLC, ITU X.25, x16+x12+x5+1
polynomial
     * Polynomial generated in the case of highest order first: Gm=0x11021; polynomial generated in the
case of lowest order first: Gm=0x8408. In this example, highest order first is used.
     */
    private static int[] crc16_ccitt_table = { 0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50a5, 0x60c6, 0x70e7,
            0x8108, 0x9129, 0xa14a, 0xb16b, 0xc18c, 0xd1ad, 0xe1ce, 0xf1ef, 0x1231, 0x0210, 0x3273, 0x2252,
            0x52b5, 0x4294, 0x72f7, 0x62d6, 0x9339, 0x8318, 0xb37b, 0xa35a, 0xd3bd, 0xc39c, 0xf3ff, 0xe3de,
            0x2462, 0x3443, 0x0420, 0x1401, 0x64e6, 0x74c7, 0x44a4, 0x5485, 0xa56a, 0xb54b, 0x8528, 0x9509,
            0xe5ee, 0xf5cf, 0xc5ac, 0xd58d, 0x3653, 0x2672, 0x1611, 0x0630, 0x76d7, 0x66f6, 0x5695, 0x46b4,
            0xb75b, 0xa77a, 0x9719, 0x8738, 0xf7df, 0xe7fe, 0xd79d, 0xc7bc, 0x48c4, 0x58e5, 0x6886, 0x78a7,
            0x0840, 0x1861, 0x2802, 0x3823, 0xc9cc, 0xd9ed, 0xe98e, 0xf9af, 0x8948, 0x9969, 0xa90a, 0xb92b,
            0x5af5, 0x4ad4, 0x7ab7, 0x6a96, 0x1a71, 0x0a50, 0x3a33, 0x2a12, 0xdbfd, 0xcbdc, 0xfbbf, 0xeb9e,
            0x9b79, 0x8b58, 0xbb3b, 0xab1a, 0x6ca6, 0x7c87, 0x4ce4, 0x5cc5, 0x2c22, 0x3c03, 0x0c60, 0x1c41,
            0xedae, 0xfd8f, 0xcdec, 0xddcd, 0xad2a, 0xbd0b, 0x8d68, 0x9d49, 0x7e97, 0x6eb6, 0x5ed5, 0x4ef4,
            0x3e13, 0x2e32, 0x1e51, 0x0e70, 0xff9f, 0xefbe, 0xdfdd, 0xcffc, 0xbf1b, 0xaf3a, 0x9f59, 0x8f78,
            0x9188, 0x81a9, 0xb1ca, 0xa1eb, 0xd10c, 0xc12d, 0xf14e, 0xe16f, 0x1080, 0x00a1, 0x30c2, 0x20e3,
            0x5004, 0x4025, 0x7046, 0x6067, 0x83b9, 0x9398, 0xa3fb, 0xb3da, 0xc33d, 0xd31c, 0xe37f, 0xf35e,
            0x02b1, 0x1290, 0x22f3, 0x32d2, 0x4235, 0x5214, 0x6277, 0x7256, 0xb5ea, 0xa5cb, 0x95a8, 0x8589,
            0xf56e, 0xe54f, 0xd52c, 0xc50d, 0x34e2, 0x24c3, 0x14a0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405,
            0xa7db, 0xb7fa, 0x8799, 0x97b8, 0xe75f, 0xf77e, 0xc71d, 0xd73c, 0x26d3, 0x36f2, 0x0691, 0x16b0,
            0x6657, 0x7676, 0x4615, 0x5634, 0xd94c, 0xc96d, 0xf90e, 0xe92f, 0x99c8, 0x89e9, 0xb98a, 0xa9ab,
            0x5844, 0x4865, 0x7806, 0x6827, 0x18c0, 0x08e1, 0x3882, 0x28a3, 0xcb7d, 0xdb5c, 0xeb3f, 0xfb1e,
            0x8bf9, 0x9bd8, 0xabbb, 0xbb9a, 0x4a75, 0x5a54, 0x6a37, 0x7a16, 0x0af1, 0x1ad0, 0x2ab3, 0x3a92,
            0xfd2e, 0xed0f, 0xdd6c, 0xcd4d, 0xbdaa, 0xad8b, 0x9de8, 0x8dc9, 0x7c26, 0x6c07, 0x5c64, 0x4c45,
            0x3ca2, 0x2c83, 0x1ce0, 0x0cc1, 0xef1f, 0xff3e, 0xcf5d, 0xdf7c, 0xaf9b, 0xbfba, 0x8fd9, 0x9ff8,
            0x6e17, 0x7e36, 0x4e55, 0x5e74, 0x2e93, 0x3eb2, 0x0ed1, 0x1ef0 };

    /**
     *
     * @param reg_init
     *            initial value during the CRC
     * @param message
     *            check code
     * @return
     */
    private static int do_crc(int reg_init, byte[] message) {
        int crc_reg = reg_init;
        for (int i = 0; i < message.length; i++) {
            crc_reg = (crc_reg >> 8) ^ crc16_ccitt_table[(crc_reg ^ message[i]) & 0xff];
        }
        return crc_reg;
    }

    /**
     * Generate a CRC code based on the data.
```



```
     *
     * @param message
     *            byte data
     *
     * @return int verification code
     */
    public static int do_crc(byte[] message) {
        // The initial value of the CRC starts from 0x0000.
        int crc_reg = 0x0000;
        return do_crc(crc_reg, message);
    }
}
```

Code example using the C-based CRC16 algorithm:

```
/**
* CCITT standard CRC16(1021) remainder table CRC16-CCITT ISO HDLC, ITU X.25, x16+x12+x5+1 polynomial
* Polynomial generated in the case of highest order first: Gm=0x11021; polynomial generated in the case of lowest order first: Gm=0x8408. In this example, highest order first is used.
*/
const unsigned short crc16_table[256] = {
    0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50a5, 0x60c6, 0x70e7,
    0x8108, 0x9129, 0xa14a, 0xb16b, 0xc18c, 0xd1ad, 0xe1ce, 0xf1ef,
    0x1231, 0x0210, 0x3273, 0x2252, 0x52b5, 0x4294, 0x72f7, 0x62d6,
    0x9339, 0x8318, 0xb37b, 0xa35a, 0xd3bd, 0xc39c, 0xf3ff, 0xe3de,
    0x2462, 0x3443, 0x0420, 0x1401, 0x64e6, 0x74c7, 0x44a4, 0x5485,
    0xa56a, 0xb54b, 0x8528, 0x9509, 0xe5ee, 0xf5cf, 0xc5ac, 0xd58d,
    0x3653, 0x2672, 0x1611, 0x0630, 0x76d7, 0x66f6, 0x5695, 0x46b4,
    0xb75b, 0xa77a, 0x9719, 0x8738, 0xf7df, 0xe7fe, 0xd79d, 0xc7bc,
    0x48c4, 0x58e5, 0x6886, 0x78a7, 0x0840, 0x1861, 0x2802, 0x3823,
    0xc9cc, 0xd9ed, 0xe98e, 0xf9af, 0x8948, 0x9969, 0xa90a, 0xb92b,
    0x5af5, 0x4ad4, 0x7ab7, 0x6a96, 0x1a71, 0x0a50, 0x3a33, 0x2a12,
    0xdbfd, 0xcbdc, 0xfbbf, 0xeb9e, 0x9b79, 0x8b58, 0xbb3b, 0xab1a,
    0x6ca6, 0x7c87, 0x4ce4, 0x5cc5, 0x2c22, 0x3c03, 0x0c60, 0x1c41,
    0xedae, 0xfd8f, 0xcdec, 0xddcd, 0xad2a, 0xbd0b, 0x8d68, 0x9d49,
    0x7e97, 0x6eb6, 0x5ed5, 0x4ef4, 0x3e13, 0x2e32, 0x1e51, 0x0e70,
    0xff9f, 0xefbe, 0xdfdd, 0xcffc, 0xbf1b, 0xaf3a, 0x9f59, 0x8f78,
    0x9188, 0x81a9, 0xb1ca, 0xa1eb, 0xd10c, 0xc12d, 0xf14e, 0xe16f,
    0x1080, 0x00a1, 0x30c2, 0x20e3, 0x5004, 0x4025, 0x7046, 0x6067,
    0x83b9, 0x9398, 0xa3fb, 0xb3da, 0xc33d, 0xd31c, 0xe37f, 0xf35e,
    0x02b1, 0x1290, 0x22f3, 0x32d2, 0x4235, 0x5214, 0x6277, 0x7256,
    0xb5ea, 0xa5cb, 0x95a8, 0x8589, 0xf56e, 0xe54f, 0xd52c, 0xc50d,
    0x34e2, 0x24c3, 0x14a0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405,
    0xa7db, 0xb7fa, 0x8799, 0x97b8, 0xe75f, 0xf77e, 0xc71d, 0xd73c,
    0x26d3, 0x36f2, 0x0691, 0x16b0, 0x6657, 0x7676, 0x4615, 0x5634,
    0xd94c, 0xc96d, 0xf90e, 0xe92f, 0x99c8, 0x89e9, 0xb98a, 0xa9ab,
    0x5844, 0x4865, 0x7806, 0x6827, 0x18c0, 0x08e1, 0x3882, 0x28a3,
    0xcb7d, 0xdb5c, 0xeb3f, 0xfb1e, 0x8bf9, 0x9bd8, 0xabbb, 0xbb9a,
    0x4a75, 0x5a54, 0x6a37, 0x7a16, 0x0af1, 0x1ad0, 0x2ab3, 0x3a92,
    0xfd2e, 0xed0f, 0xdd6c, 0xcd4d, 0xbdaa, 0xad8b, 0x9de8, 0x8dc9,
    0x7c26, 0x6c07, 0x5c64, 0x4c45, 0x3ca2, 0x2c83, 0x1ce0, 0x0cc1,
    0xef1f, 0xff3e, 0xcf5d, 0xdf7c, 0xaf9b, 0xbfba, 0x8fd9, 0x9ff8,
    0x6e17, 0x7e36, 0x4e55, 0x5e74, 0x2e93, 0x3eb2, 0x0ed1, 0x1ef0
};

int do_crc(int reg_init, byte* data, int length)
{
    int cnt;
    int crc_reg = reg_init;
    for (cnt = 0; cnt < length; cnt++)
    {
        crc_reg = (crc_reg >> 8) ^ crc16_table[(crc_reg ^ *(data++)) & 0xFF];
    }
    return crc_reg;
}

int main(int argc, char **argv)
{
    // FFFE011300000000 is represented by a byte array.
    byte message[8] = {0xFF,0xFE,0x01,0x13,0x00,0x00,0x00,0x00};
```



```
    // The initial value of the CRC starts from 0x0000.
    int a = do_crc(0x0000, message, 8);
    printf("a ==> %x\n", a);
}
```

FAQ

OTA Upgrades

Best Practices

Performing OTA Firmware Upgrade for MQTT Devices

3.9.2 PCP Introduction

The PCP protocol stipulates the communication content and format between the IoT platform and devices.

PCP runs at the application layer for device upgrade.

Communication Method

1. PCP runs at the application layer. LwM2M, CoAP, MQTT, or other non-streaming protocols can be used at the underlying layer.
2. PCP messages are not allocated with independent ports and are independent from protocols at the underlying layer. To differentiate PCP messages from device service messages, 0XFFFE is used as the start bytes of the PCP messages, and the first two bytes of the service messages cannot be 0XFFFE. For details, see **PCP Message Identification**.
3. PCP uses a question-and-answer communication mode. All request messages have a response message.

Message Structure

Field	Type	Description
Start ID	WORD	**Start ID**: The value is fixed at 0XFFFE.
Version	BYTE	The four most significant bits are reserved. The four least significant bits indicate the protocol version. Currently, the version is **1**.

Field	Type	Description
Message code	BYTE	Type of the request exchanged between the platform and device. The message code of a response is the same as that of the request. The following message codes have been defined: • 0-18: reserved • 19: device version query • 20: software package notification • 21: software package download • 22: download result reporting • 23: upgrade execution • 24: upgrade result reporting • 25-127: reserved
Check code	WORD	CRC16 check value calculated from the start ID to the last byte of the data zone. Before the calculation, this field is set to **0**. The result is then written to the field after the CRC16 calculation. **NOTE** CRC16 algorithm: CRC16/CCITT x16+x12+x5+1
Data zone length	WORD	Length of the data zone.
Data zone	BYTE[n]	Variable length, which is defined by each instruction. For details, see the definitions of the request and response corresponding to each instruction.

Data Type

Data Type	Description
BYTE	Unsigned 1-byte integer
WORD	Unsigned 2-byte integer
DWORD	Unsigned 4-byte integer
BYTE[n]	Hexadecimal number of *n* bytes
STRING	String

NOTE

PCP uses the network sequence to transmit WORD and DWORD data.

Device Version Query

Request

Direction: from the platform to a device

Field	Data Type	Description
No data zone		

Response

Direction: from a device to the platform

Field	Data Type	Description
Result code	BYTE	**0X00**: The processing was successful.
Current version	BYTE[16]	The version is described using ASCII characters. If there are not enough available digits, 0X00 is appended.

NOTE

- The platform determines whether the device needs to be upgraded based on the version. If it does, the platform sends a request to upgrade the device.
- If the response times out, the platform stops the upgrade task.

Software Package Notification

Request

Direction: from the platform to a device

Field	Data Type	Description
Target version	BYTE[16]	The version is described using ASCII characters. If there are not enough available digits, 0X00 is appended.
Upgrade package segment size	WORD	Size of each segment.

Field	Data Type	Description
Number of upgrade package segments	WORD	Number of upgrade package segments
Check code	WORD	The value is fixed at **0000**.

Response

Direction: from a device to the platform

Field	Data Type	Description
Result code	BYTE	**0X00**: The upgrade is allowed. **0X01**: The device is in use. **0X02**: The signal is weak. **0X03**: The latest version is in use. **0X04**: The battery power is low. **0X05**: The remaining space is insufficient. **0X09**: The memory is insufficient. **0X7F**: An internal error has occurred.

NOTE

- If the upgrade is not allowed by the device, the platform stops the upgrade task.
- If the response times out, and the request for the upgrade package is not received, the platform stops the upgrade task.

Software Package Requesting

Request

Direction: from a device to the platform

Field	Data Type	Description
Target version	BYTE[16]	The version is described using ASCII characters. If there are not enough available digits, 0X00 is appended.
Segment sequence number	WORD	Sequence number of the requested segment. The value starts from 0. The total number of segments is obtained by rounding up the result of the software package size divided by the segment size. The device can save the received segments and request for the missing segments next time. Resumable download is supported.

Response

Direction: from the platform to a device

Field	Data Type	Description
Result code	BYTE	**0X00**: The processing was successful. **0X80**: The upgrade task does not exist. **0X81**: The specified segment does not exist.
Segment sequence number	WORD	Sequence number of a returned segment.
Segment data	BYTE[n]	Content of the segment. **n** indicates the segment size. If the result code is not 0, this field is not included.

Download Result Reporting

Request

Direction: from a device to the platform

Field	Data Type	Description
Download status	BYTE	**0X00**: The upgrade package has been downloaded. **0X05**: The remaining space is insufficient. **0X06**: The download timed out. **0X07**: The upgrade package failed to be verified. **0X08**: The upgrade package is not supported.

Response

Direction: from the platform to a device

Field	Data Type	Description
Result code	BYTE	**0X00**: The processing was successful. **0X80**: The upgrade task does not exist.

Upgrade Execution

Request

Direction: from the platform to a device

Field	Data Type	Description
No data zone		

Response

Direction: from a device to the platform

Field	Data Type	Description
Result code	BYTE	**0X00**: The processing was successful. **0X01**: The device is in use. **0X04**: The battery power is low. **0X05**: The remaining space is insufficient. **0X09**: The memory is insufficient.

Upgrade Result Reporting

Request

Direction: from a device to the platform

Field	Data Type	Description
Result code	BYTE	**0X00**: The upgrade was successful. **0X01**: The device is in use. **0X04**: The battery power is low. **0X05**: The remaining space is insufficient. **0X09**: The memory is insufficient. **0X0A**: The upgrade package failed to be installed. **0X7F**: An internal error has occurred.
Current version	BYTE[16]	Current version of the device.

Response

Direction: from the platform to a device

Field	Data Type	Description
Result code	BYTE	**0X00**: The processing was successful. **0X80**: The upgrade task does not exist.

PCP Message Identification

PCP messages and device service messages share the same port and URL. When receiving a message from the device, the platform performs the following steps to determine whether the message is a PCP message or a service message:

1. Checks whether the device supports software upgrades (defined by **omCapability.upgradeCapability** in the product model). If the device does not support software upgrades, the message is considered to be a service message.
2. Checks whether the software upgrade protocol is PCP. If the protocol is not PCP, the message is considered to be a service message.
3. Checks whether the first two bytes of the message are 0XFFFE. If the bytes are not 0XFFFE, the message is considered to be a service message.
4. Checks whether the version is valid. If the version is invalid, the message is considered as a service message.
5. Checks whether the message code is valid. If the message code is invalid, the message is considered as a service message.
6. Checks whether the check code is correct. If the check code is incorrect, the service message is considered to be a service message.
7. Checks whether the length of the data zone is correct. If the length is incorrect, the message is considered to be a service message.
8. If all the preceding check items are passed, the message is considered as a PCP message.

NOTE

The start bytes of a service message cannot be 0XFFFE.

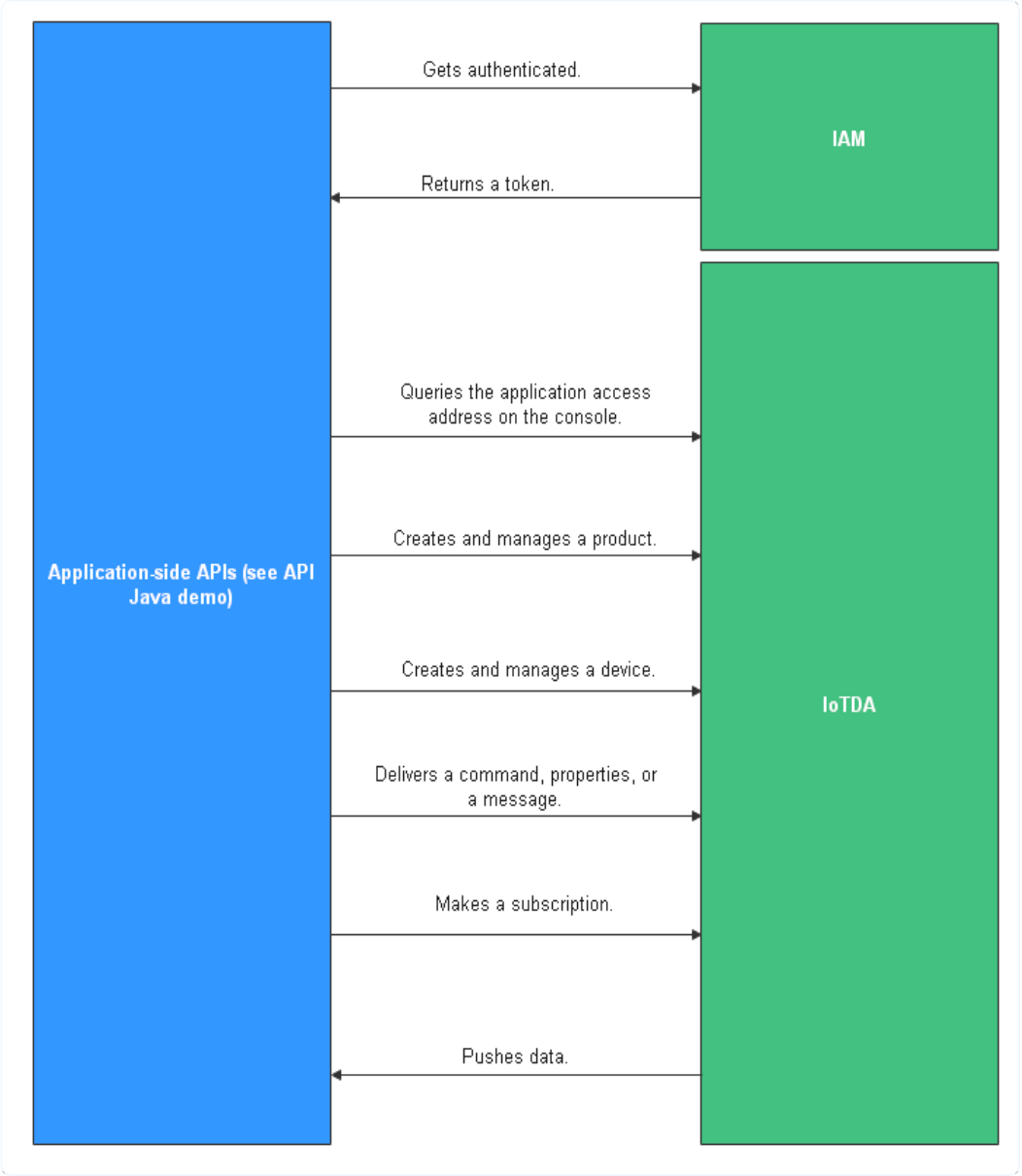
4 Development on the Application Side

4.1 API Usage Guide

The IoT platform provides a variety of APIs to make application development easier and more efficient. You can call these open APIs to quickly integrate platform functions, such as management of products, devices, subscriptions, commands, and rules.

NOTE

The application needs to be authenticated by the IAM service. To obtain a token, see **Debugging the API Obtaining the Token for an IAM User**.



Application Development Resources

The platform provides a wealth of application-side APIs to ease application development. Applications can call these APIs to implement services such as secure access, device management, data collection, and command delivery.

Resource Package	Description	Download Link
Application API Java Demo	You can call **application-side APIs** to experience service functions and service processes.	**API Java Demo**

Resource Package	Description	Download Link
Application Java SDK	You can use Java methods to call **application-side APIs** to communicate with the platform. For details, see **Java SDK**.	**Application Java SDK**
Application .NET SDK	You can use .NET methods to call **application-side APIs** to communicate with the platform. For details, see **.NET SDK**.	**Application .NET SDK**
Application Python SDK	You can use Python methods to call **application-side APIs** to communicate with the platform. For details, see **Python SDK**.	**Application Python SDK**
Application Go SDK	You can use Go methods to call **application-side APIs** to communicate with the platform. For details, see **Go SDK**.	**Application Go SDK**
Application Node.js SDK	You can use Node.js methods to call **application-side APIs** to communicate with the platform. For details, see **Node.js SDK**.	**Application Node.js SDK**
Application PHP SDK	You can use PHP methods to call **application-side APIs** to communicate with the platform. For details, see **PHP SDK**.	**Application PHP SDK**

API Introduction

API Group	Scenario
Product management	Used to manage product models that have been imported to the platform. A product model defines the capabilities or features of all devices under a product.

API Group	Scenario
Device management	Used by applications to manage devices, including basic device details and device data.
Device message	Used by applications to transparently transmit messages to devices.
Device command	Used by applications to deliver commands to devices for control. A product model defines commands that the platform can deliver to devices.
Device property	Used by applications to deliver properties to devices. A product model defines properties that the platform can deliver to devices.
AMQP queue management	Used to create, delete, and view queues. AMQP queues can receive messages through AMQP clients after subscribing to rules.
Access credential management	Used for authentication when long connections are established using protocols such as AMQP and MQTTS.
Data transfer rule management APIs and **device linkage rule APIs**	Used by applications to set rules to implement service linkage or forward data to other Huawei Cloud services. Device linkage and data forwarding rules are available. • A device linkage rule consists of triggers and actions. When the configured trigger is met, the corresponding action is triggered, for example, delivering commands, sending notifications, reporting alarms, and clearing alarms. • For a data forwarding rule, you need to set forwarding data, set forwarding targets, and start the rule. Data can be forwarded to Data Ingestion Service (DIS), Distributed Message Service (DMS) for Kafka, Object Storage Service (OBS), ROMA Connect, third-party application (HTTP push), and AMQP message queue.
Subscription management APIs	Used by applications to subscribe to resources provided by the platform. If the subscribed resources change, the platform notifies the applications of the change.

API Group	Scenario
Device shadow APIs	Used by applications to operate and manage the device shadow. A device shadow is a file used to store and retrieve the status of a device. • Each device has only one device shadow, which is uniquely identified by the device ID. • The device shadow saves only the latest data reported by the device and the desired data set by an application. • You can use the device shadow to query and set the device status regardless of whether the device is online.
Device group management APIs	Used by applications to manage device groups, including group details and device members in a group.
Tag management APIs	Used by applications to bind tags to or unbind tags from resources. Currently, only devices support tags.
Resource space management	Used by applications to manage resource spaces, including adding, deleting, modifying, and querying resource spaces.
Batch task APIs	Used by applications to perform batch operations on devices connected to the platform. • Supported batch operations: upgrading software and firmware, creating, deleting, updating, freezing, and unfreezing devices, creating synchronous and asynchronous commands, creating messages, and setting device shadow. • Up to 10 unfinished tasks of the same type is allowed for a user. When the maximum number is reached, new tasks cannot be created.
Device CA certificate management APIs	Used by applications to manage device CA certificates, including uploading, verifying, and querying certificates. The platform supports device access authentication using certificates.
OTA upgrade package management	Used by applications to operate and manage upgrade packages, including creating, querying, and deleting upgrade packages.
Broadcast message	Used by applications to broadcast messages to all online devices that subscribe to specified topics.

API Group	Scenario
Device tunnel management	Used for data transmission between applications and devices.
Data stack policy management	Used by applications to manage stack policies, including creating, querying, modifying, and deleting stack policies.
Data flow control policy management	Used by applications to manage flow control policies, including creating, querying, modifying, and deleting flow control policies.

FAQ

Application Integration

Message Communications

Subscription and Push

How Does IoTDA Obtain Device Data?

What Should I Do If I Want to Call an API But Have No Permissions to Do So as an IAM User?

4.2 Debugging Using Postman

Overview

Postman is a visual editing tool for building and testing API requests. It provides an easy-to-use UI to send HTTP requests, including GET, PUT, POST, and DELETE requests, and modify parameters in HTTP requests. Postman also returns response to your requests.

To fully understand APIs, refer to **API Reference on the Application Side**. The Postman Collection is already available, in which the structure of API call requests are ready for use.

This topic uses Postman as an example to describe how to debug the following APIs when the application simulator connects to the IoT platform using HTTPS:

- **Obtaining the Token of an IAM User**
- **Listing Projects Accessible to an IAM User**
- **Creating a Product**
- **Querying a Product**

- **Creating a Device**
- **Querying a Device**

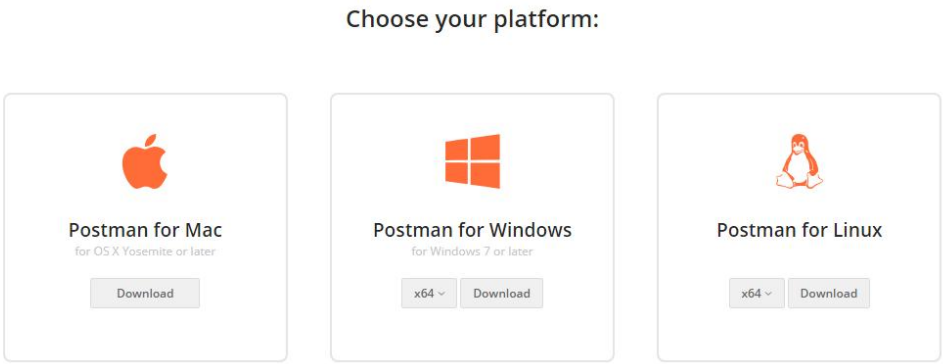
Prerequisites

- You have installed Postman. If Postman is not installed, install it by following the instructions provided in **Installing and Configuring Postman**.
- You have downloaded the **Collection**.
- You have developed a **product model** and a **codec** on the **console**.

Installing and Configuring Postman

Step 1 Install Postman.

1. Visit the **Postman website**, and download and install the latest version of Postman (64-bit) for Windows.



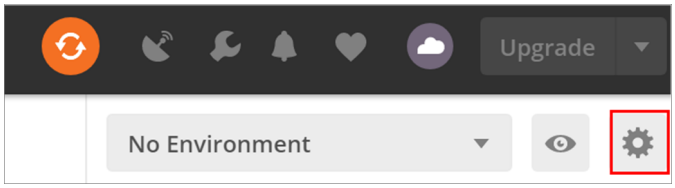
NOTE

- Postman requires the **.NET Framework 4.5** component.
- To ensure successful API calls, you are advised to download **the latest version of Postman (32-bit) for Windows**.

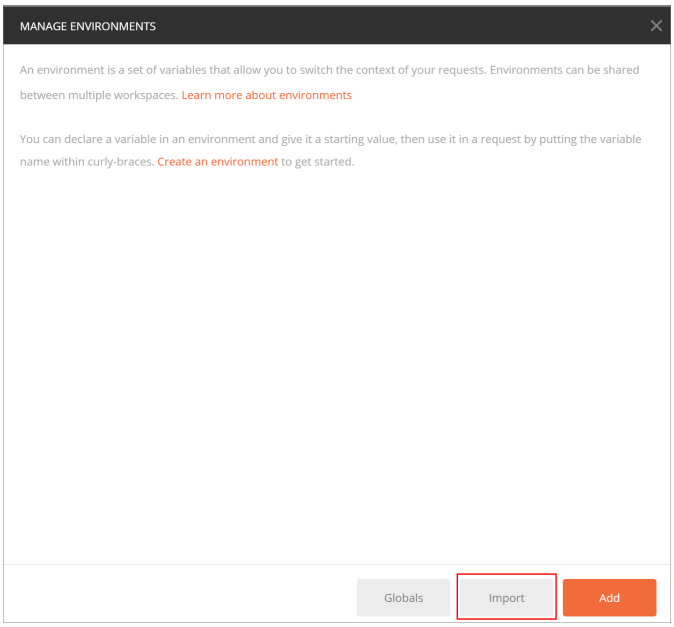
2. Enter the email address, username, and password to register Postman.

Step 2 Import the Postman environment variables.

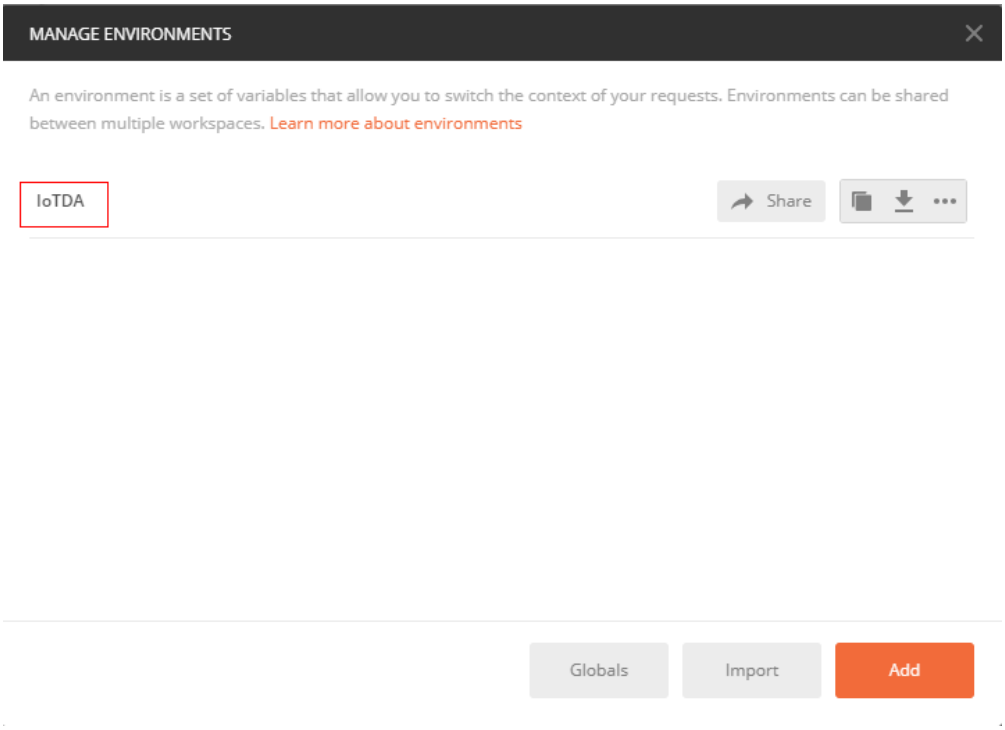
1. Click  in the upper right corner to open the **MANAGE ENVIRONMENTS** window.



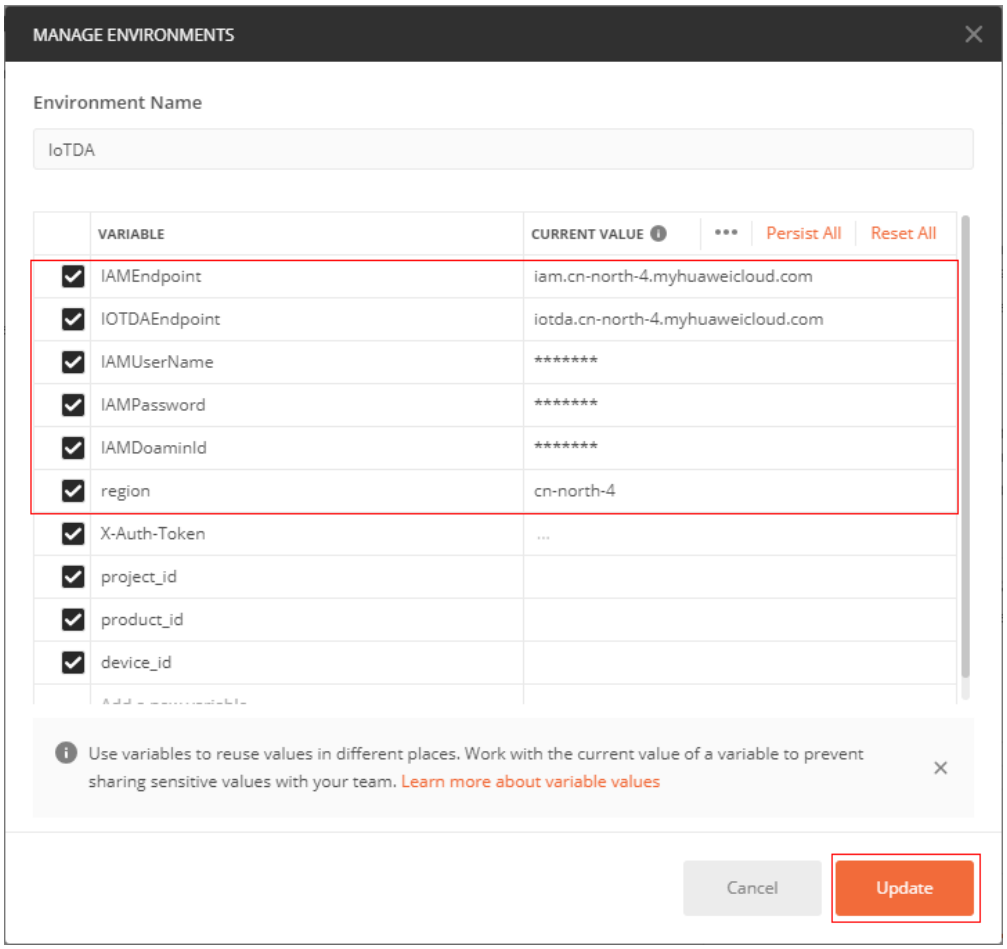
2. Click **Import**. On the page displayed, click **Select File** to import the **IoTDA.postman_environment.json** file (obtained after the **Collection** package is decompressed).



3. Click the **IoTDA** environment imported.



4. Configure parameters based on the following table.

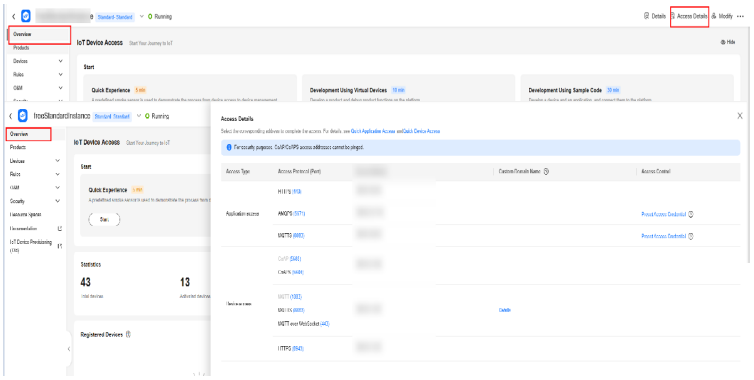


Parameter	Description
IAMEndpoint	IAM endpoint. For details, see **Regions and Endpoints**.
IoTDAEndpoint	IoTDA endpoint. For details, see **Step 2.5**.
IAMUserName	IAM username, which can be obtained from the **My Credentials** page.
IAMPassword	Password for logging in to Huawei Cloud.
IAMDoaminId	Account name, which can be obtained from the **My Credentials** page.
region	Region where IoTDA is enabled.

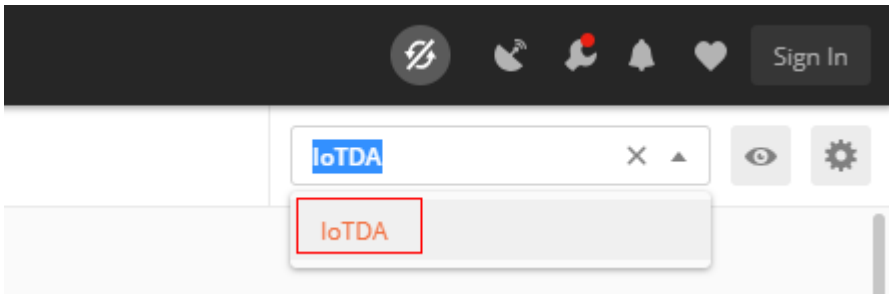
5. Obtain IoTDA endpoints.

   Log in to the console. In the navigation pane, choose **Overview**. Click **Access Details** in the **Instance Information** area. Select the access address based on the access type and protocol.

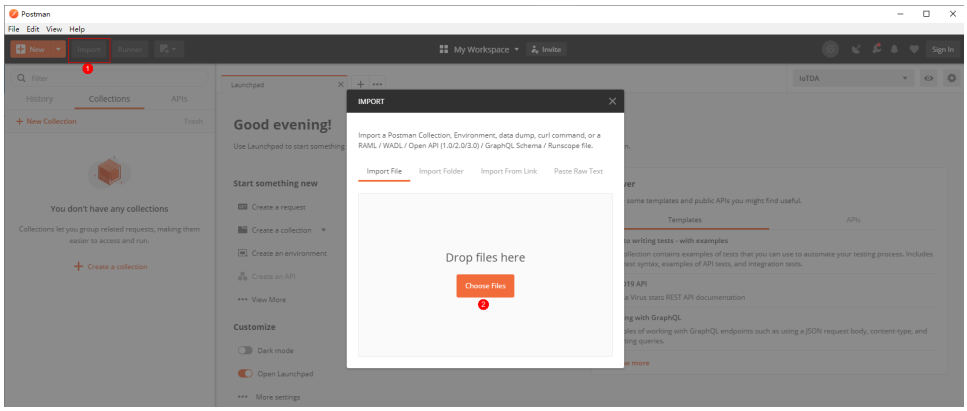
Figure 4-1 Obtaining access information



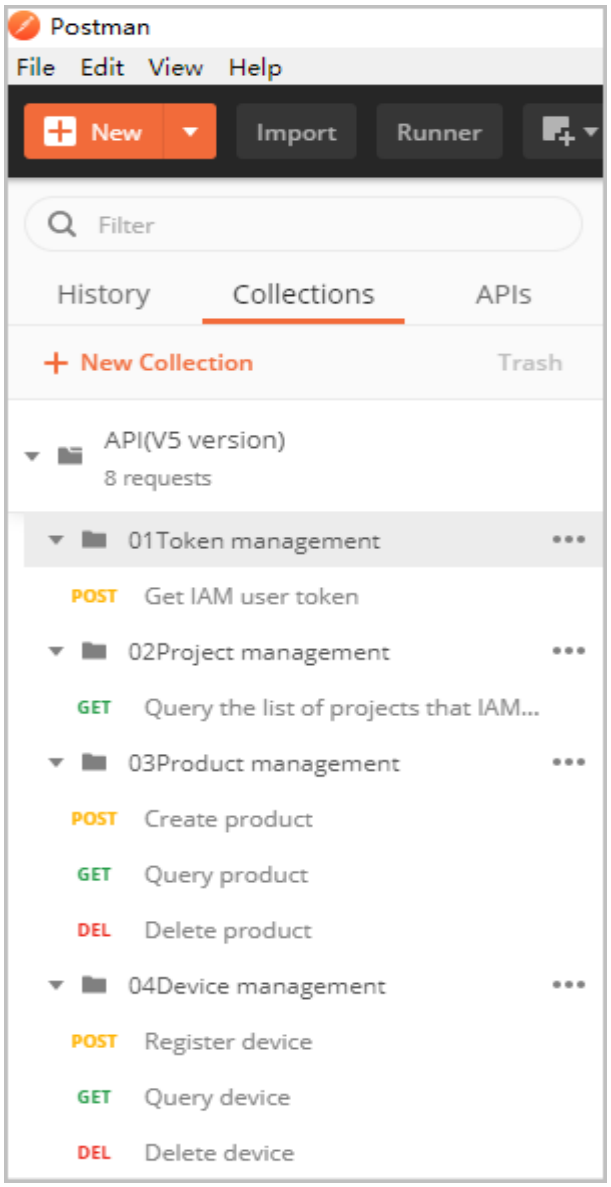
6. Return to the home page and set the environment variable to the imported IoTDA.



Step 3 Click **Import** in the upper left corner and click **Choose Files** to import the **API call (V5).postman_collection.json** file.



After the file is uploaded, the dialog box shown in the following figure is displayed.



----End

Debugging the API Obtaining the Token for an IAM User

Before using platform APIs, an application must call the API **Obtaining the Token of an IAM User** for authentication. After the authentication is successful, Huawei Cloud returns **X-Subject-Token**.

To call this API, the application constructs an HTTP request. An example request is as follows:

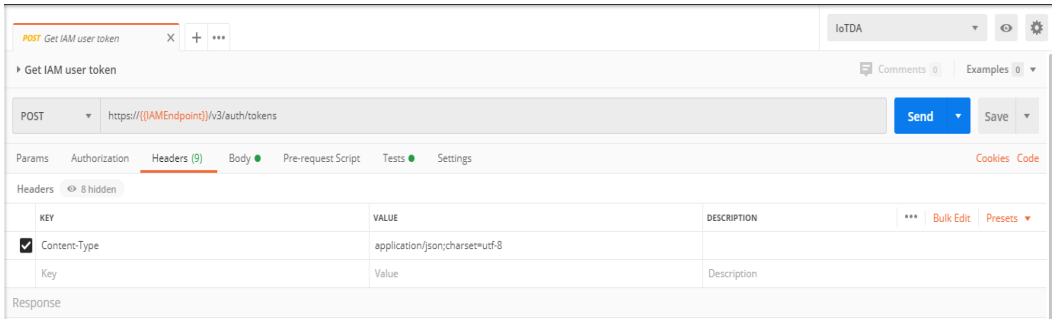
```
POST https://iam.cn-north-4.myhuaweicloud.com/v3/auth/tokens
Content-Type: application/json

{
    "auth": {
        "identity": {
            "methods": [
                "password"
            ],
```



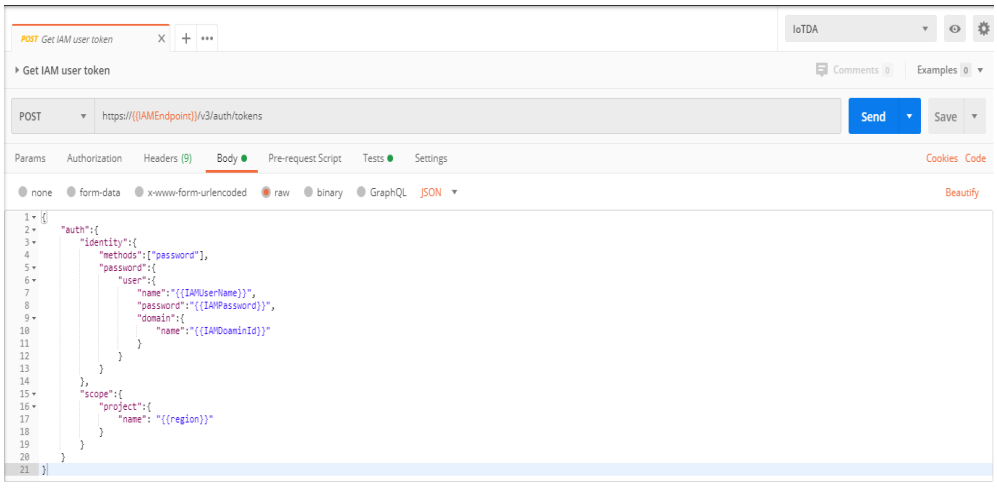
```
            "password": {
                "user": {
                    "name": "username",
                    "password": "********",
                    "domain": {
                        "name": "domainname"
                    }
                }
            }
        },
        "scope": {
            "project": {
                "name": "xxxxxxxx"
            }
        }
    }
}
```

Debug the API by following the instructions provided in **Obtaining the Token of an IAM User**.

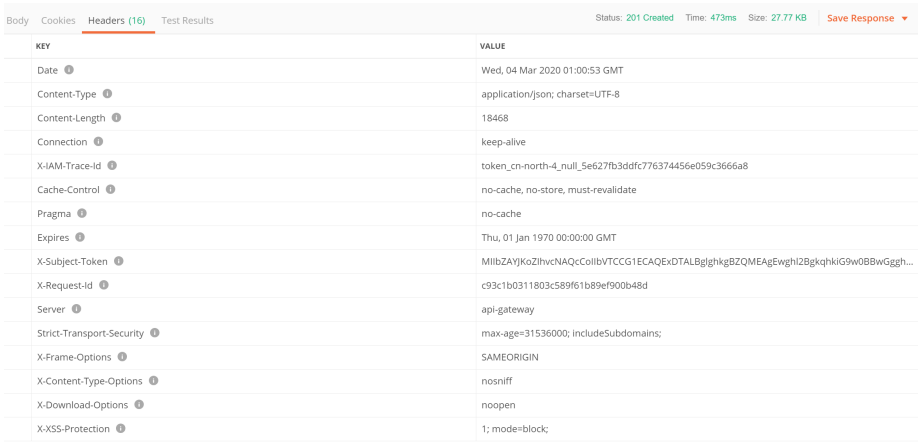
Step 1 Configure the HTTP method, URL, and headers of the API.



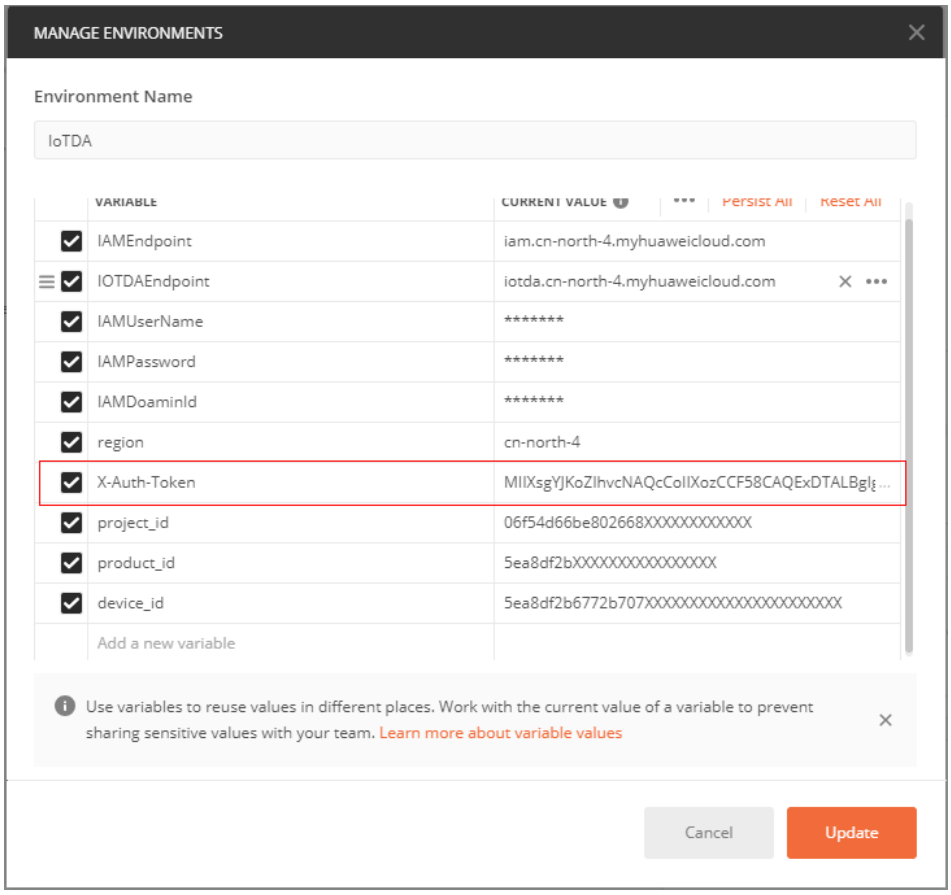
Step 2 Configure the body of the API.



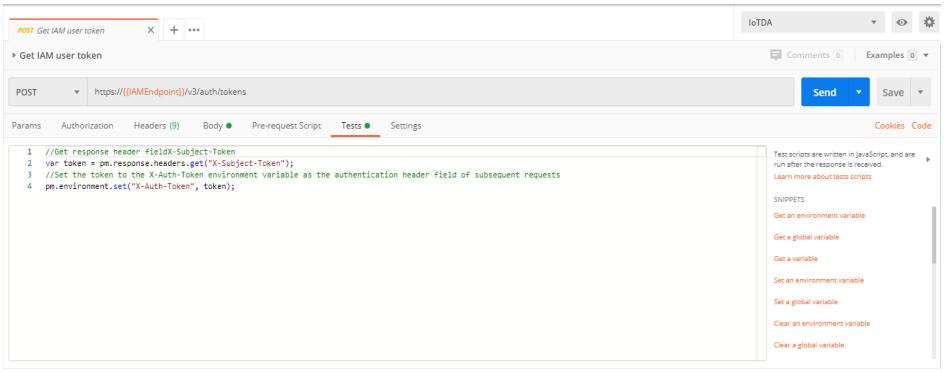
Step 3 Click **Send**. The returned code and response are displayed in the lower part of the page.



Step 4 Use the returned **X-Subject-Token** value in the header field to update **X-Auth-Token** in the IoTDA environment so that it can be used in other API calls. If the token expires, the **Authentication** API must be called again to obtain a new token.



The **X-Auth-Token** parameter is automatically updated in Postman. You do not need to manually update it.



----End

Debugging the API Listing Projects Accessible to an IAM User

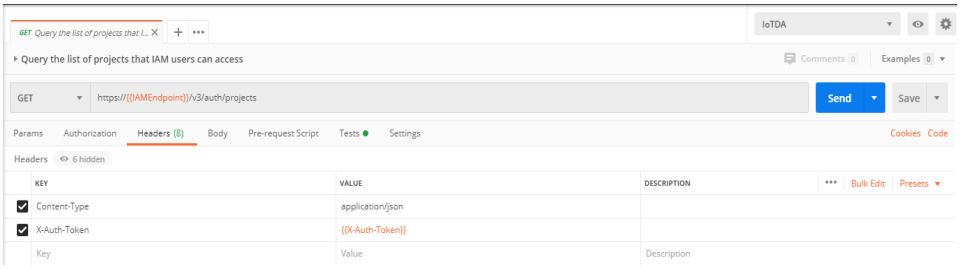
Before accessing platform APIs, the application must call the API **Listing Projects Accessible to an IAM User** to obtain the project ID of the user.

To call this API, the application constructs an HTTP request. An example request is as follows:

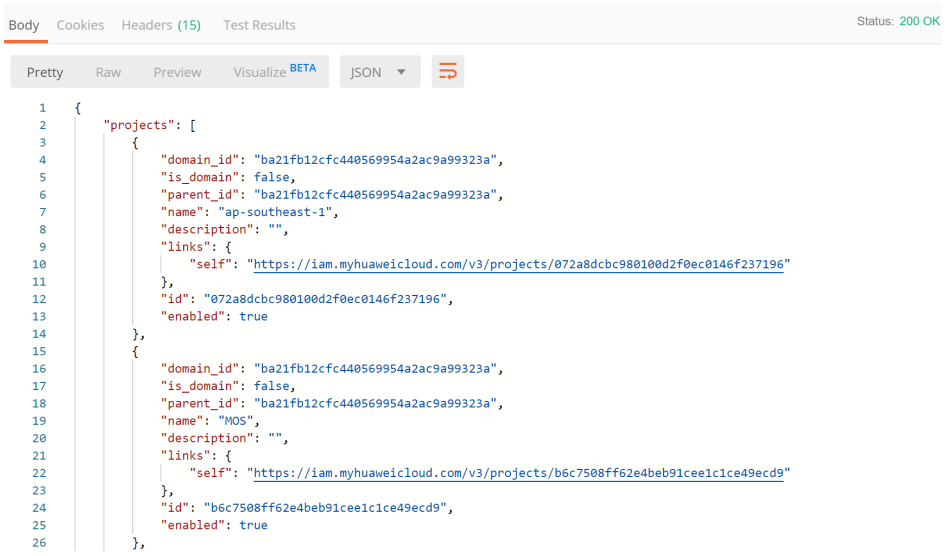
```
GET https://iam.cn-north-4.myhuaweicloud.com/v3/auth/projects
Content-Type: application/json
X-Auth-Token: ********
```

Debug the API by following the instructions provided in **Listing Projects Accessible to an IAM User**.

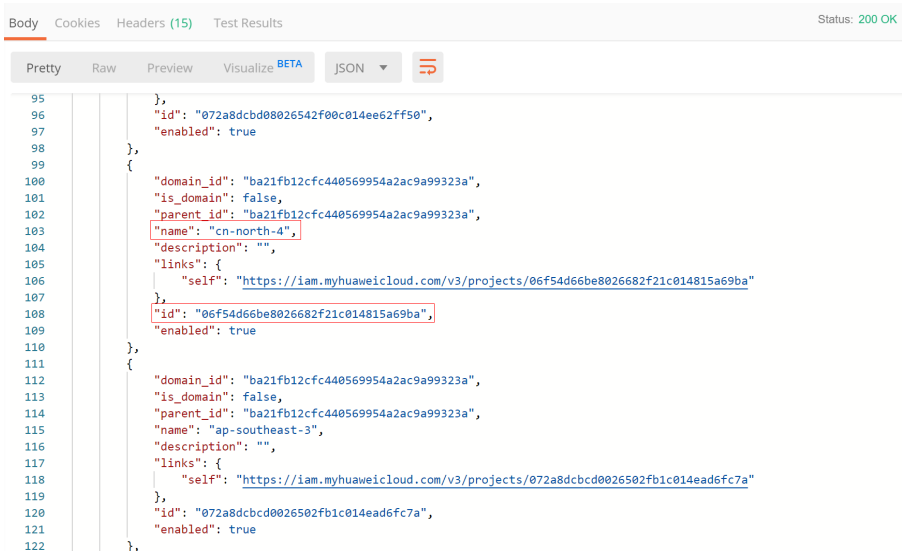
Step 1 Configure the HTTP method, URL, and headers of the API.

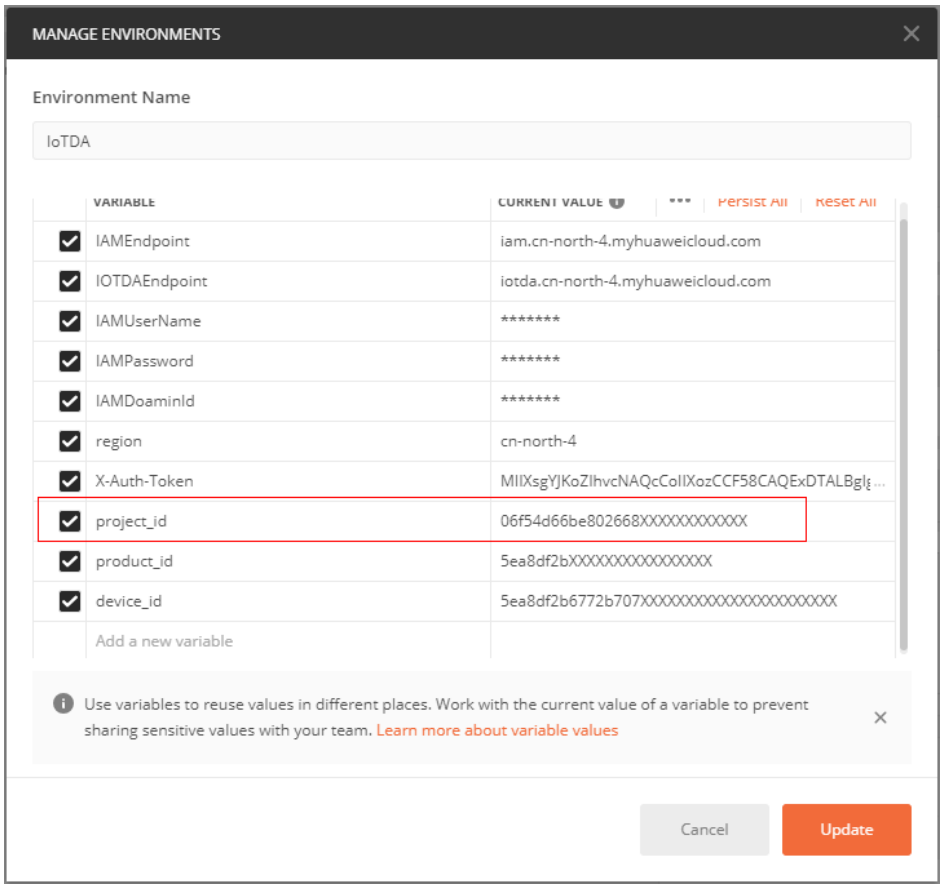


Step 2 Click **Send**. The returned code and response are displayed in the lower part of the page.

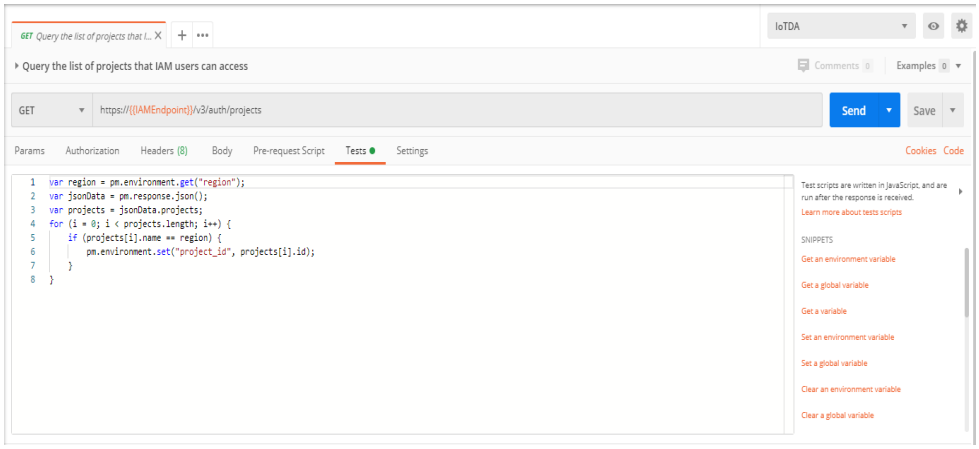


Step 3 The returned body contains a list of projects. Search for the item whose **name** is the same as the value of **region** in the IoTDA environment, and use the **id** value to update **project_id** in the IoTDA environment so that it can be used in other API calls.





In this example, the **project_id** parameter is automatically updated in Postman. You do not need to manually update it.



----End

Debugging the API Creating a Product

Before connecting a device to the platform, an application must call the API **Creating a Product**. The product created will be used during device registration.

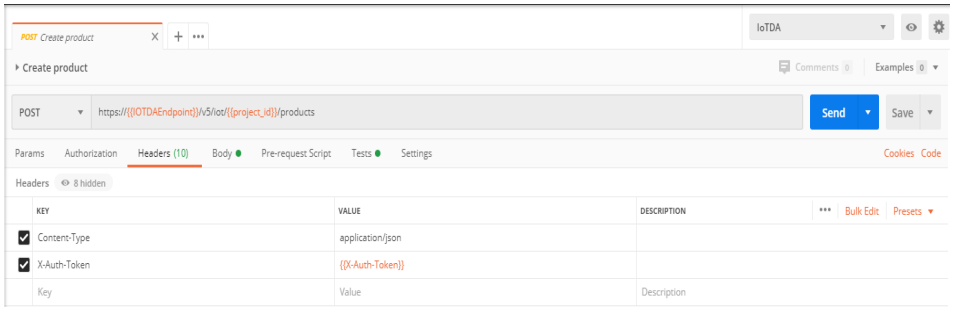
To call this API, the application constructs an HTTP request. An example request is as follows:


```
POST https://iotda.cn-north-4.myhuaweicloud.com/v5/iot/{project_id}/products
Content-Type: application/json
X-Auth-Token: ********

{
  "name" : "Thermometer",
  "device_type" : "Thermometer",
  "protocol_type" : "MQTT",
  "data_format" : "binary",
  "manufacturer_name" : "ABC",
  "industry" : "smartCity",
  "description" : "this is a thermometer produced by Huawei",
  "service_capabilities" : [ {
    "service_type" : "temperature",
    "service_id" : "temperature",
    "description" : "temperature",
    "properties" : [ {
      "unit" : "centigrade",
      "min" : "1",
      "method" : "R",
      "max" : "100",
      "data_type" : "decimal",
      "description" : "force",
      "step" : 0.1,
      "enum_list" : [ "string" ],
      "required" : true,
      "property_name" : "temperature",
      "max_length" : 100
    } ],
    "commands" : [ {
      "command_name" : "reboot",
      "responses" : [ {
        "response_name" : "ACK",
        "paras" : [ {
          "unit" : "km/h",
          "min" : "1",
          "max" : "100",
          "para_name" : "force",
          "data_type" : "string",
          "description" : "force",
          "step" : 0.1,
          "enum_list" : [ "string" ],
          "required" : false,
          "max_length" : 100
        } ]
      } ],
      "paras" : [ {
        "unit" : "km/h",
        "min" : "1",
        "max" : "100",
        "para_name" : "force",
        "data_type" : "string",
        "description" : "force",
        "step" : 0.1,
        "enum_list" : [ "string" ],
        "required" : false,
        "max_length" : 100
      } ]
    } ],
    "option" : "Mandatory"
  } ],
  "app_id" : "jeQDJQZltU8iKgFFoW060F5SGZka"
}
```

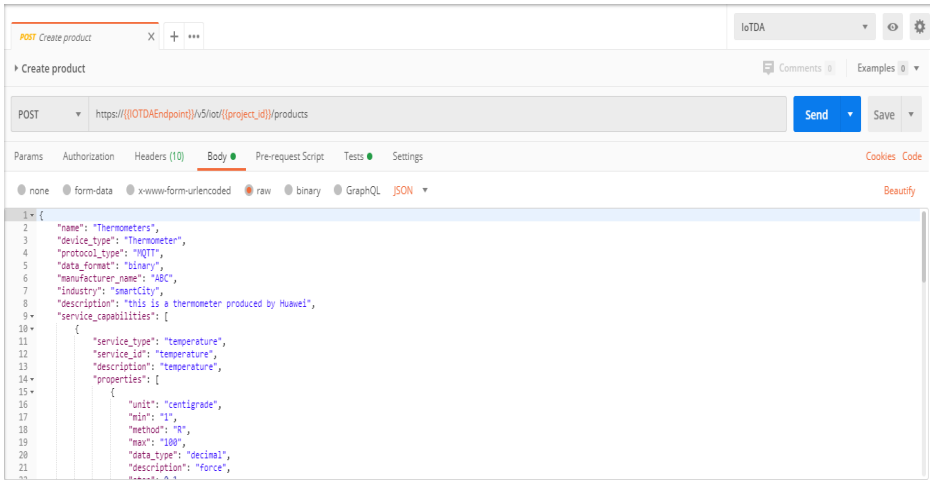
Debug the API by following the instructions provided in **Creating a Product**.

Note: Only the parameters used in the debugging example are described in the following steps.

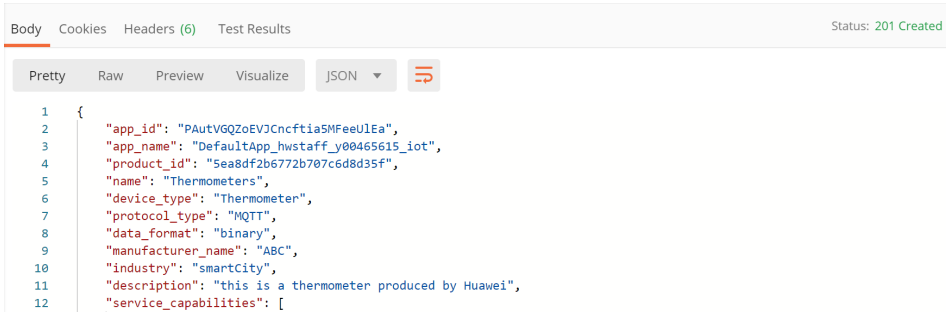
Step 1 Configure the HTTP method, URL, and headers of the API.



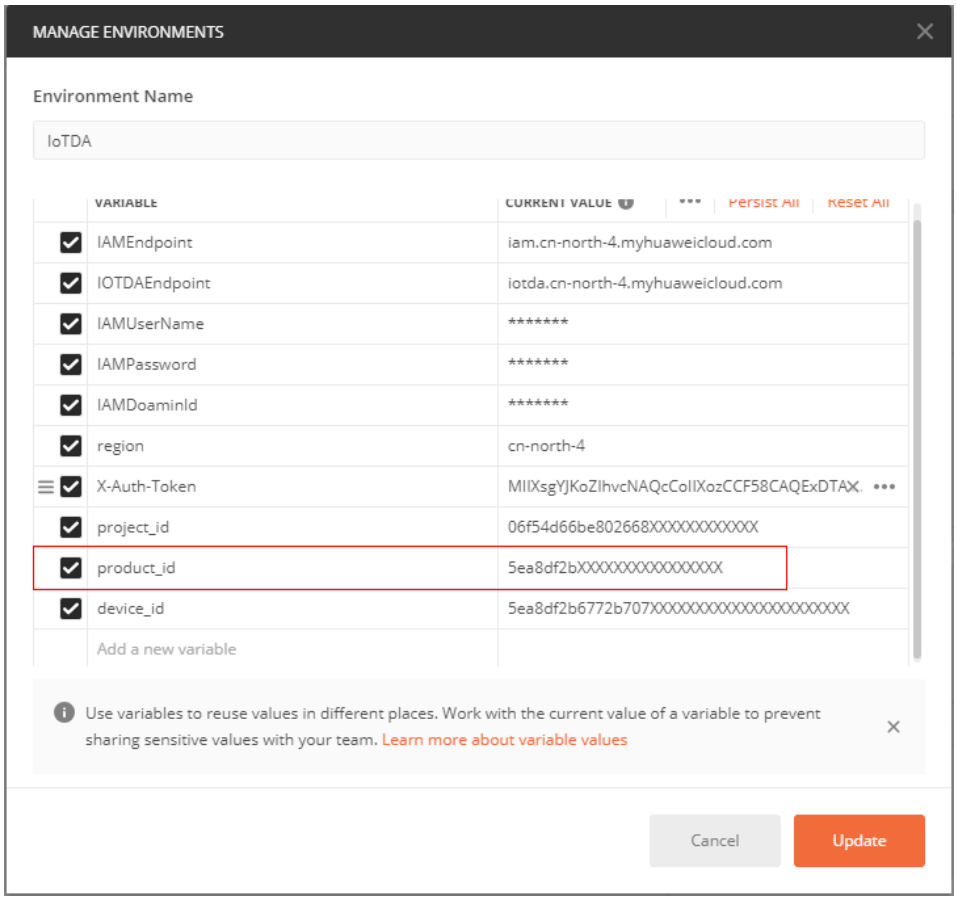
Step 2 Configure the body of the API.



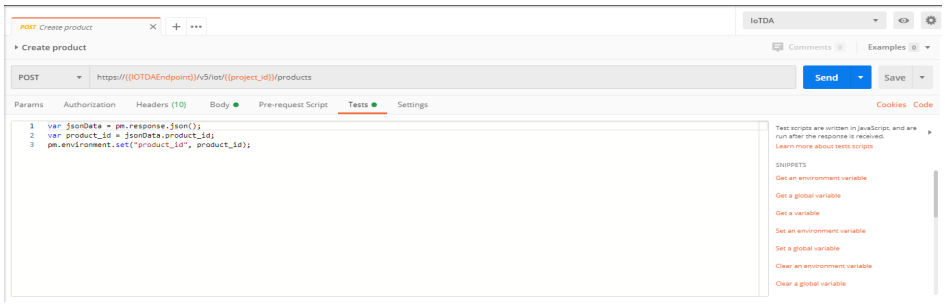
Step 3 Click **Send**. The returned code and response are displayed in the lower part of the page.



Step 4 Use the returned **product_id** value to update the **product_id** parameter in the IoTDA environment so that it can be used in other API calls.



Note: The **product_id** parameter is automatically updated in Postman. You do not need to manually update it.



----End

Debugging the API Querying a Product

An application can call the API **Querying a Product** to query details about a product.

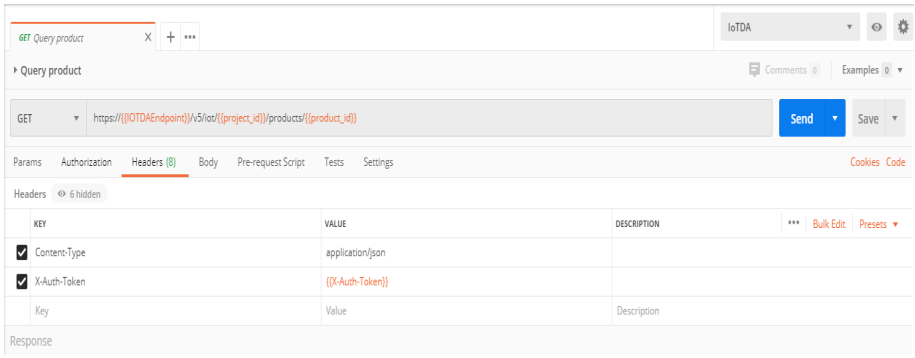
To call this API, the application constructs an HTTP request. An example request is as follows:

```
GET https://iotda.cn-north-4.myhuaweicloud.com/v5/iot/{project_id}/products/{product_id}
Content-Type: application/json
X-Auth-Token: ********
```

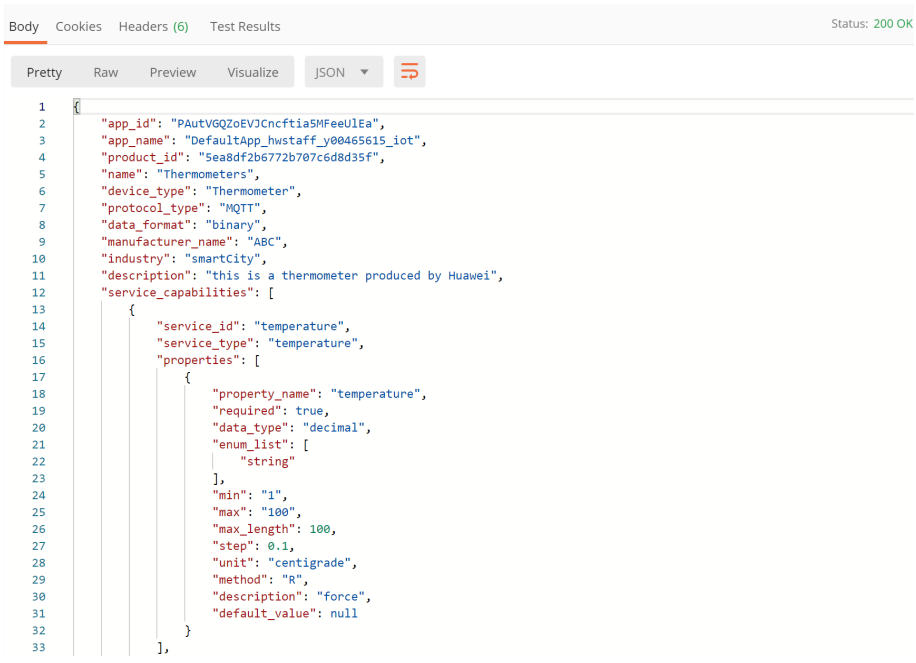
Debug the API by following the instructions provided in **Querying a Product**.

Note: Only the parameters used in the debugging example are described in the following steps.

Step 1 Configure the HTTP method, URL, and headers of the API.



Step 2 Click **Send**. The returned code and response are displayed in the lower part of the page.



----End

Debugging the API Creating a Device

Before connecting a device to the platform, an application must call the API **Registering a Device**. Then, the device can use the unique identification code to get authenticated and connect to the platform.

To call this API, the application constructs an HTTP request. An example request is as follows:

```
POST https://iotda.cn-north-4.myhuaweicloud.com/v5/iot/{project_id}/devices
Content-Type: application/json
X-Auth-Token: ********

{
  "node_id" : "ABC123456789",
```

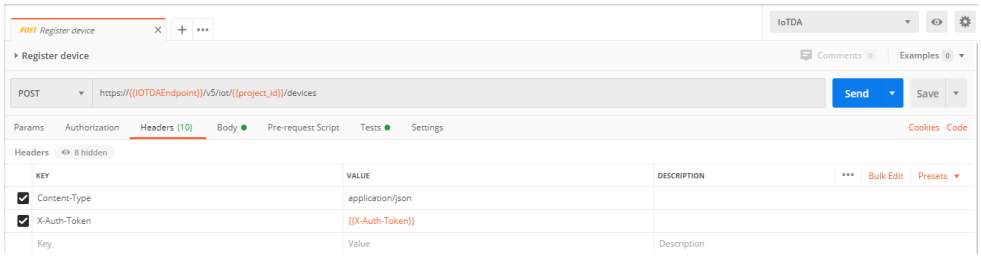


```
 "device_name" : "dianadevice",
 "product_id" : "b640f4c203b7910fc3cbd446ed437cbd",
 "auth_info" : {
   "auth_type" : "SECRET",
   "secure_access" : true,
   "fingerprint" : "********",
   "secret" : "********",
   "timeout" : 300
 },
 "description" : "water meter device"
}
```

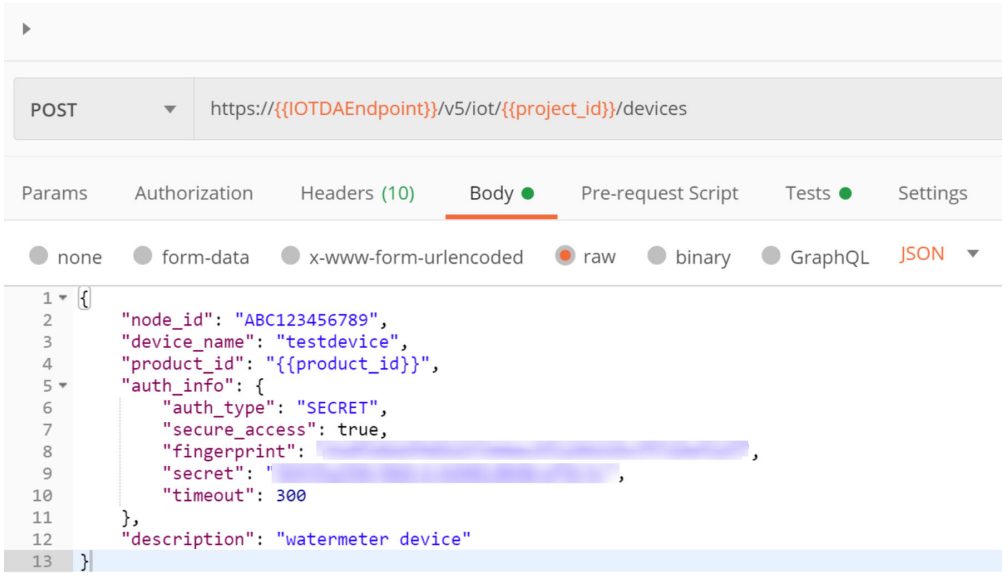
Debug the API by following the instructions provided in **Creating a Device**.

Note: Only the parameters used in the debugging example are described in the following steps.

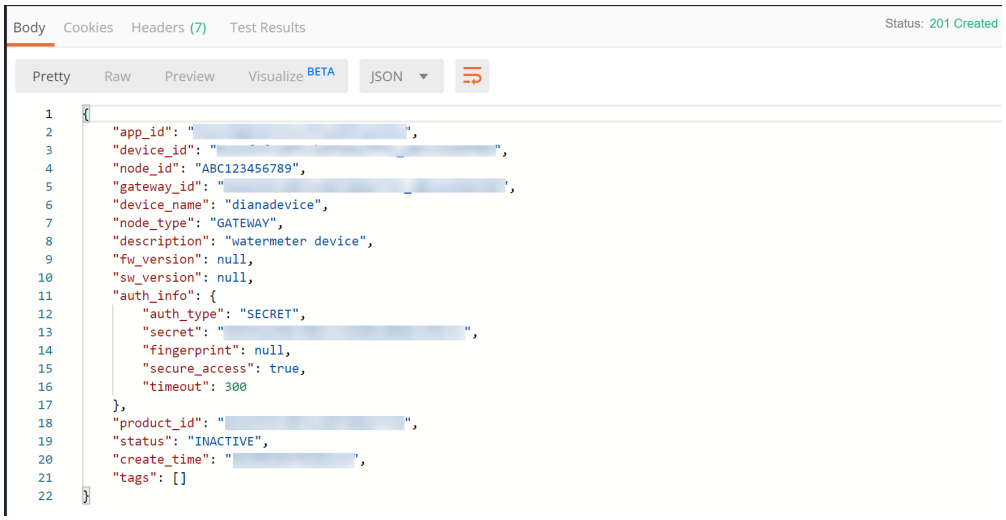
Step 1 Configure the HTTP method, URL, and headers of the API.



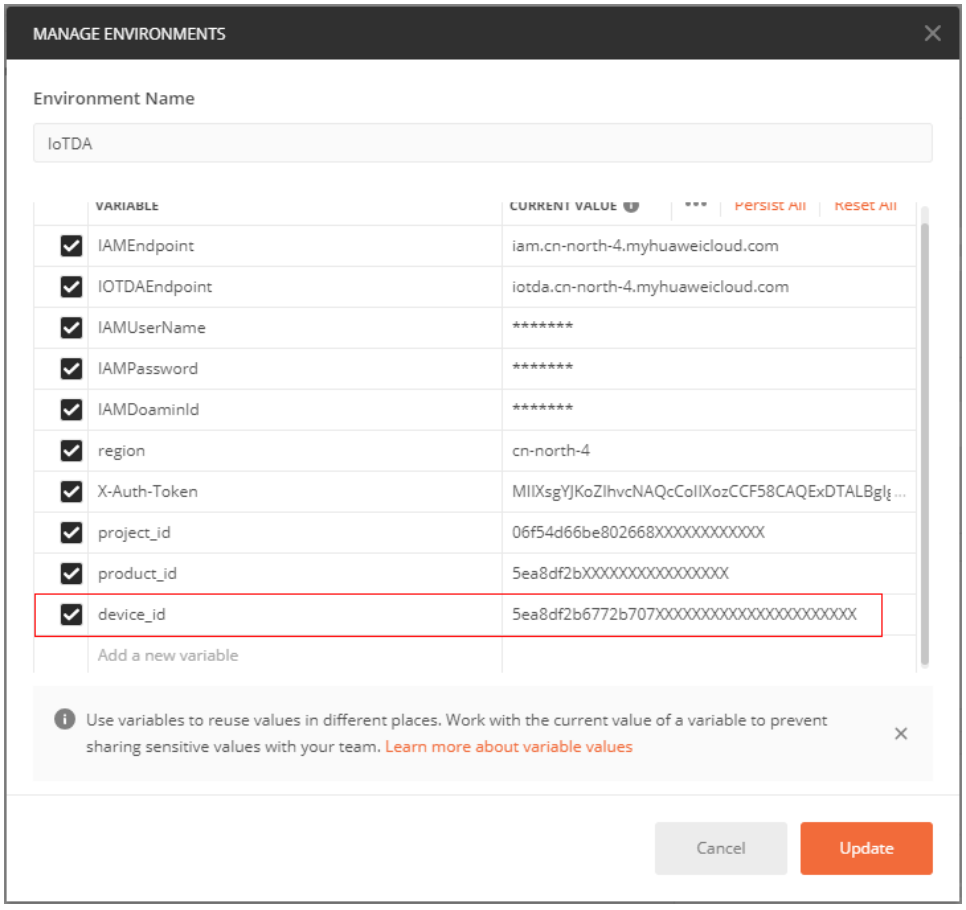
Step 2 Configure the body of the API.



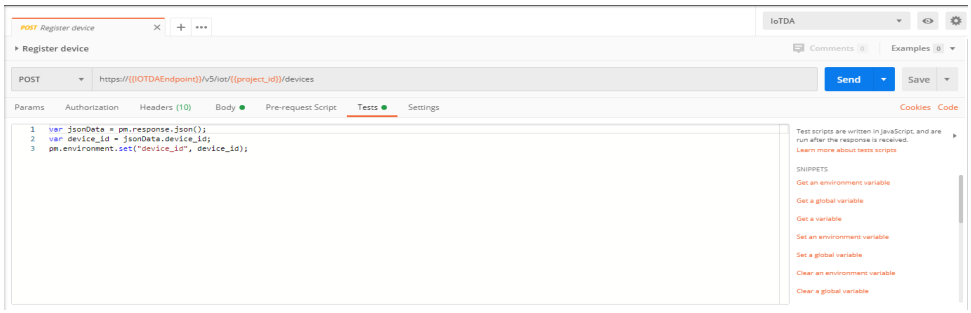
Step 3 Click **Send**. The returned code and response are displayed in the lower part of the page.



Step 4 Use the returned **device_id** value to update the **device_id** parameter in the IoTDA environment so that it can be used in other API calls.



Note: The **device_id** parameter is automatically updated in Postman. You do not need to manually update it.



----End

Debugging the API Querying a Device

An application can call the API **Querying a Device** to query details about a device registered with the platform.

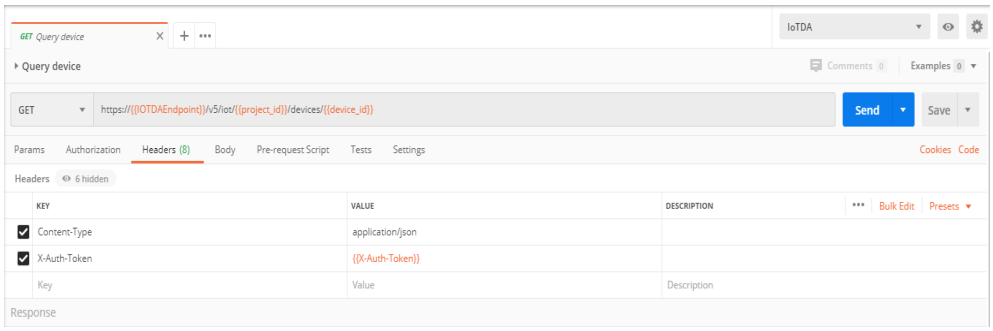
To call this API, the application constructs an HTTP request. An example request is as follows:

```
GET https://iotda.cn-north-4.myhuaweicloud.com/v5/iot/{project_id}/devices/{device_id}
Content-Type: application/json
X-Auth-Token: ********
```

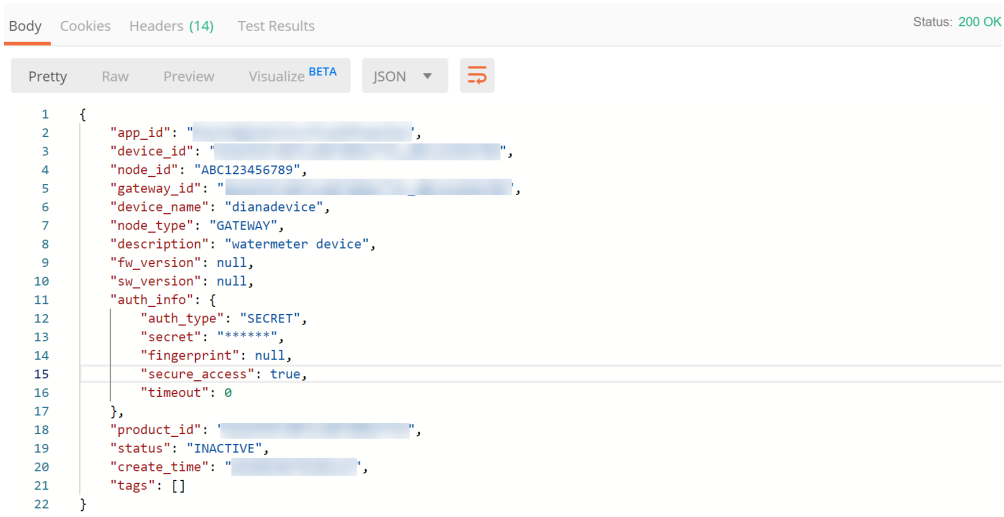
Debug the API by following the instructions provided in **Querying a Device**.

Note: Only the parameters used in the debugging example are described in the following steps.

Step 1 Configure the HTTP method, URL, and headers of the API.



Step 2 Click **Send**. The returned code and response are displayed in the lower part of the page.



----End

