IoT Device Access

Developer Guide

Issue 1.0

Date 2024-10-12

HUAWEI

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.

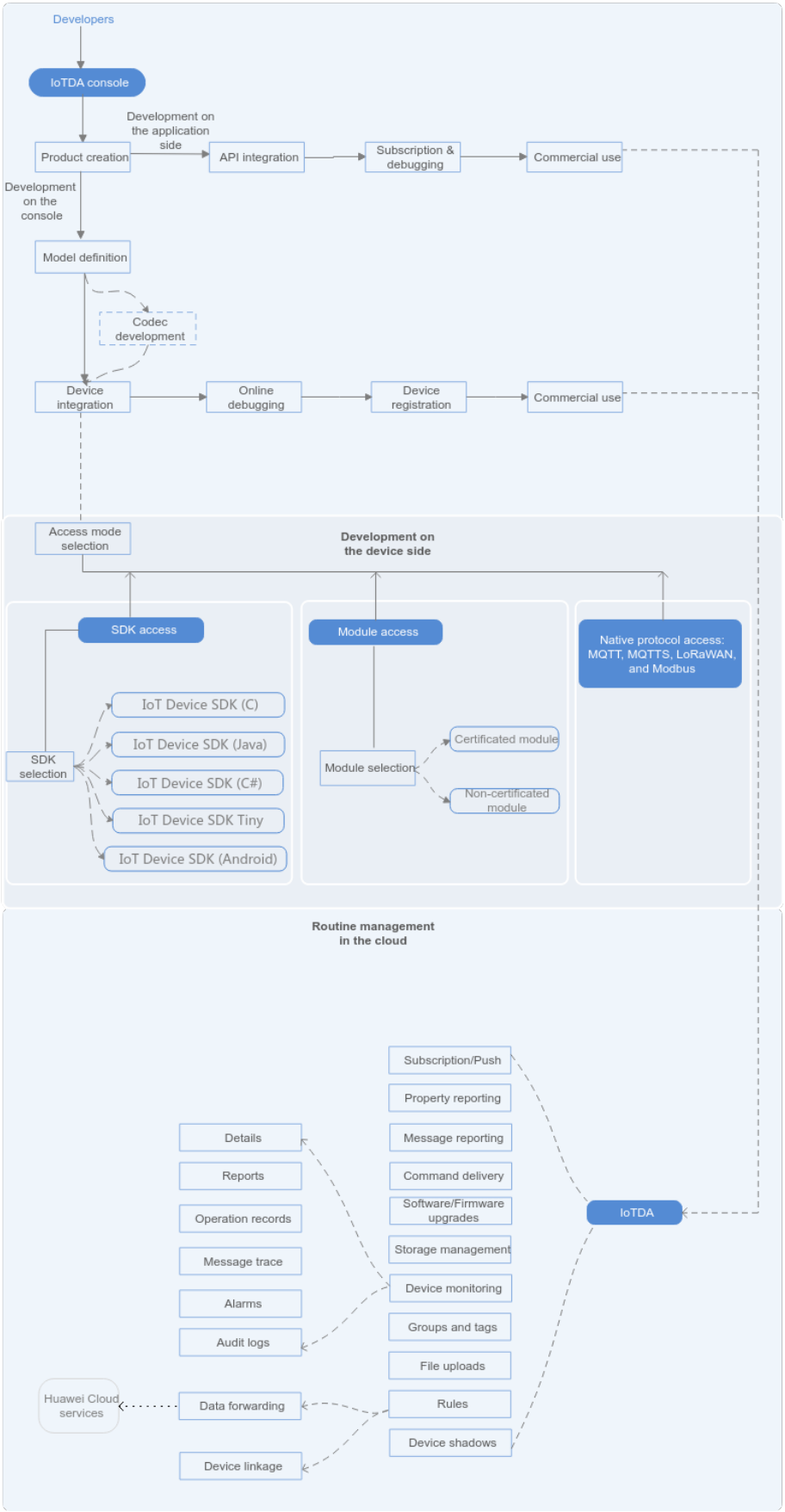
Service Process

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

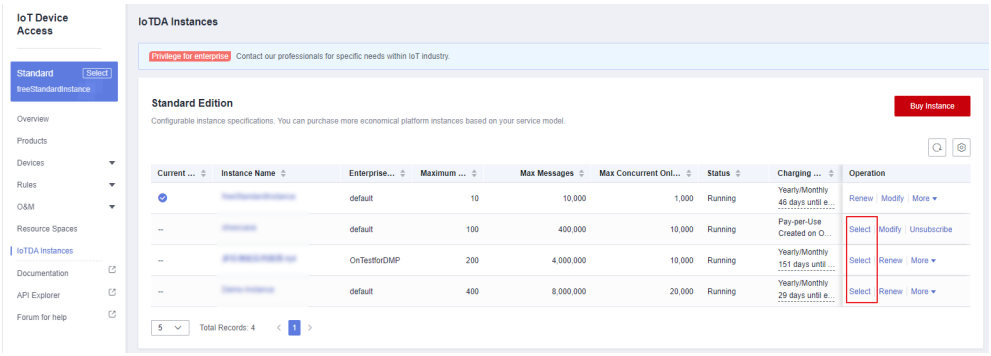


2 Obtaining Resources

Platform Connection Information

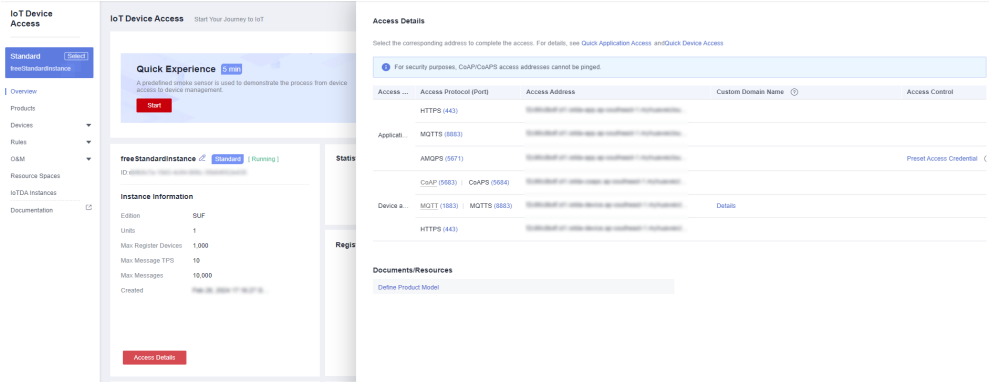
1. Log in to the **IoTDA console**. In the navigation pane, choose **IoTDA Instances**, and select an edition as required.

Figure 2-1 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 SDK (Java)	Devices can connect to the platform by integrating the IoT Device SDK (Java). The demo provides the sample code for calling SDK APIs. For details, see **IoT Device SDK (Java)**.	**IoT Device SDK (Java)**
IoT Device SDK (C)	Devices can connect to the platform by integrating the IoT Device SDK (C). The demo provides the sample code for calling SDK APIs. For details, see **IoT Device SDK (C)**.	**IoT Device SDK (C)**
IoT Device SDK (C#)	Devices can connect to the platform by integrating the IoT Device SDK (C#). The demo provides the sample code for calling SDK APIs. For details, see **IoT Device SDK (C#)**.	**IoT Device SDK (C#)**
IoT Device SDK (Android)	Devices can connect to the platform by integrating the IoT Device SDK (Android). The demo provides the sample code for calling SDK APIs. For details, see **IoT Device SDK (Android)**.	**IoT Device SDK (Android)**
IoT Device SDK (Go)	Devices can connect to the platform by integrating the IoT Device SDK (Go). The demo provides the code sample for calling the SDK APIs. For details, see **IoT Device SDK (Go) User Guide**.	**IoT Device SDK (Go)**

Resource Package	Description	Download Link
IoT Device SDK(Python)	Devices can connect to the platform by integrating the IoT Device SDK (Python). The demo provides the code sample for calling the SDK APIs. For details, see **IoT Device SDK (Python) Usage Guide**.	**IoT Device SDK(Python)**
IoT Device SDK Tiny (C)	Devices can connect to the platform by integrating the IoT Device SDK Tiny (C). The demo provides the sample code for calling SDK APIs. For details, see **IoT Device Tiny SDK (C)**.	**IoT Device SDK Tiny (C)**
Native MQTT or MQTTS access example	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)**
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**

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

Resource Package	Description	Download Link
Application Go SDK	You can use Go methods to call **application-side APIs** to communicate with the platform. For details, see **Go SDK**.	**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**.	**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**.	**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 cannot be used to verify server certificates after their expiration dates. Replace these certificates before expiration dates 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, and AF-Johannesburg	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**

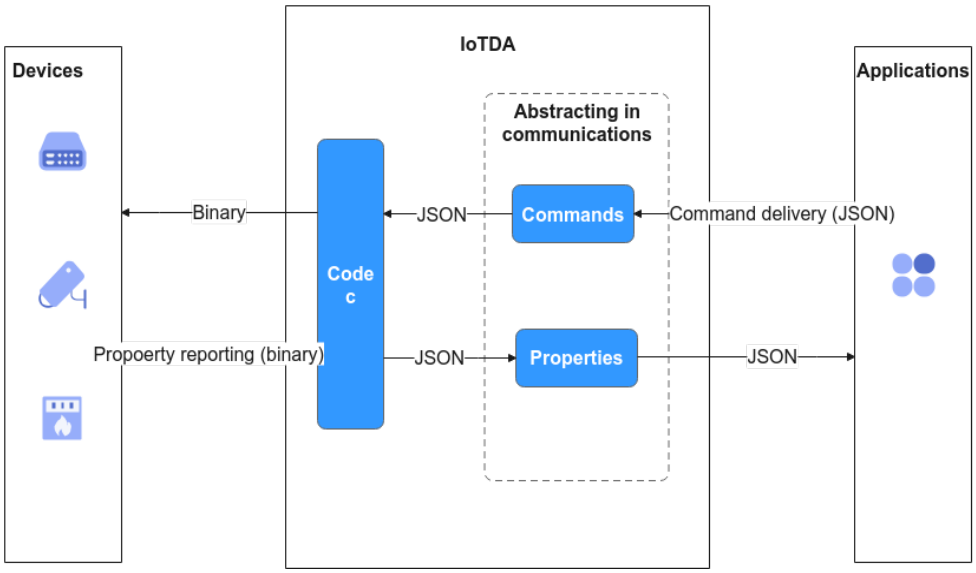
3 Product Development

3.1 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.

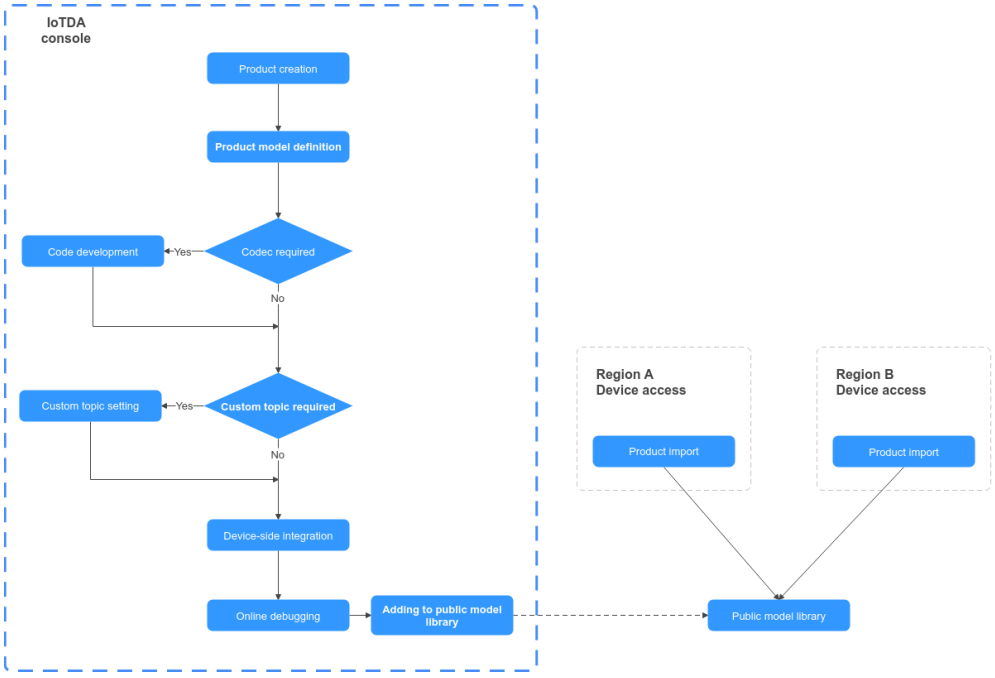
Figure 3-1 Codec usage process



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 3-2 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 (profiles), 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: The IoTDA console 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.

3.2 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**.

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: MQTT is used by devices to access the platform. The data format can be binary or JSON. If the binary format is used, the codec must be deployed. • LwM2M over CoAP: LwM2M/CoAP is used only by NB-IoT devices with limited resources (including storage and power consumption). The data format is binary. The codec must be deployed to interact with the platform. • HTTPS is a secure communication protocol based on HTTP and encrypted using SSL. IoTDA supports communication through HTTPS. • Modbus: Modbus is used by devices to access the platform. Devices that use the Modbus protocol to connect to IoT edge nodes are called indirectly connected devices. For details about the differences between directly connected devices and indirectly connected devices, see **Gateways and Child Devices**. • HTTP (TLS encryption), ONVIF, OPC UA, OPC DA, and other: 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.
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.

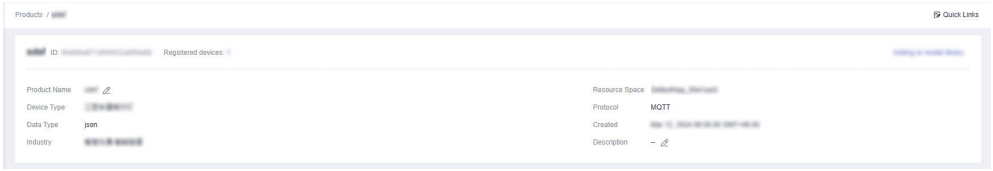
You can click **Delete** to delete a product that is no longer used. After the product is deleted, its resources such as the product models and codecs will be cleared. Exercise caution when deleting a product.

----End

Follow-Up Procedure

1. In the product list, click the name of a product to access its details. 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 3-3 Product details

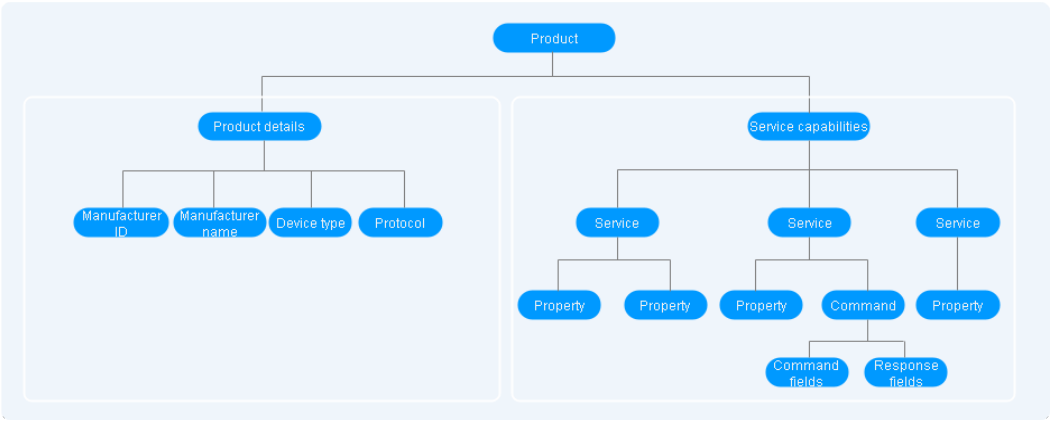


2. On the product details page, you can **develop a product model**, **develop a codec**, **perform online debugging**, and **customize topics**.

3.3 Developing a Product Model

3.3.1 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 color or any 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.

Service Name	Description
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.

3.3.2 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**.

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 **Model Definition** tab page, click **Customize Model** to add 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).

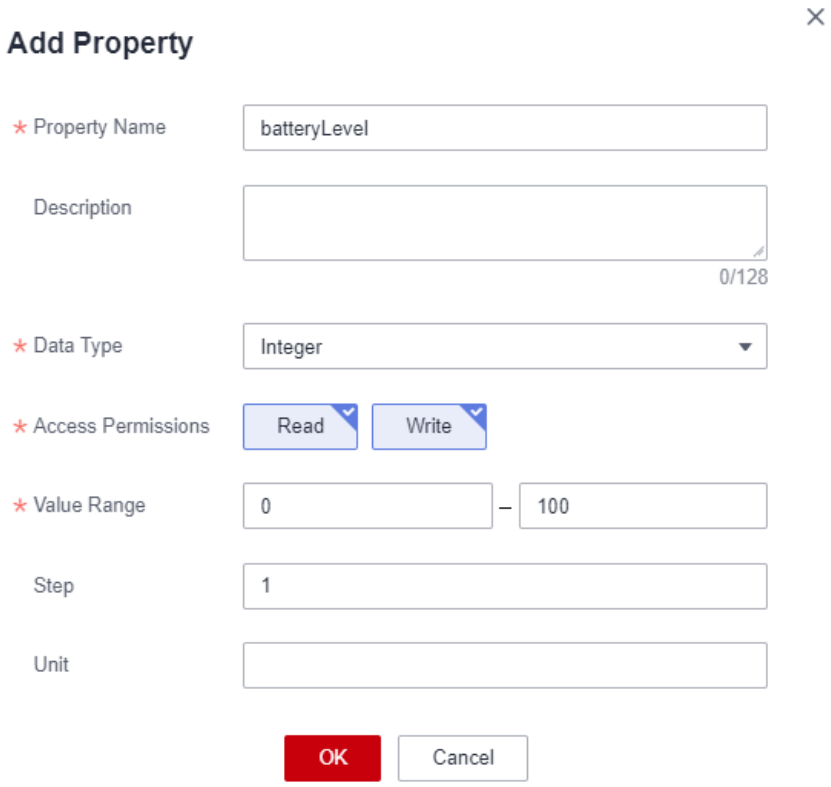
After the service is added, define the properties and commands in the **Add Service** area. A service can contain properties and/or commands. Configure the properties and commands based on your requirements.

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**.

Parameter	Description
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 3-4 Adding a property

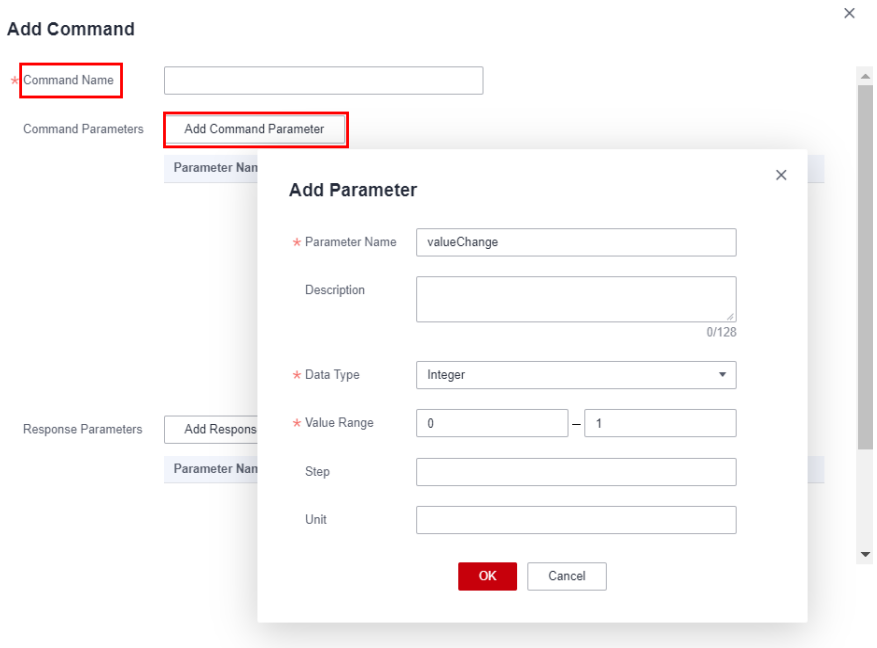


Step 6 Click **Add Command**. In the dialog box displayed, set command parameters.

- **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 Name	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	

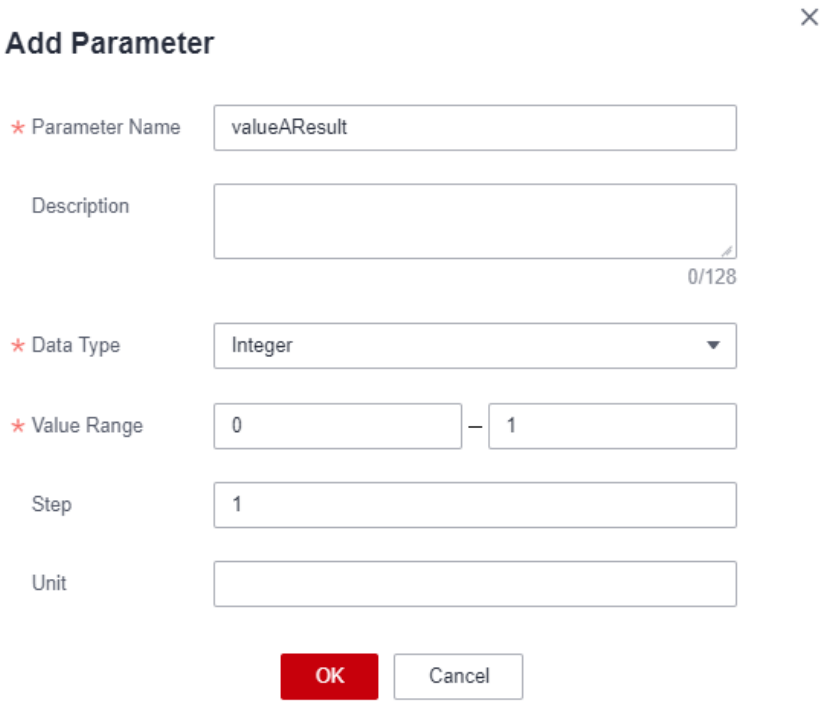
Figure 3-5 Adding a command



- 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 Name	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 3-6 Adding a response parameter

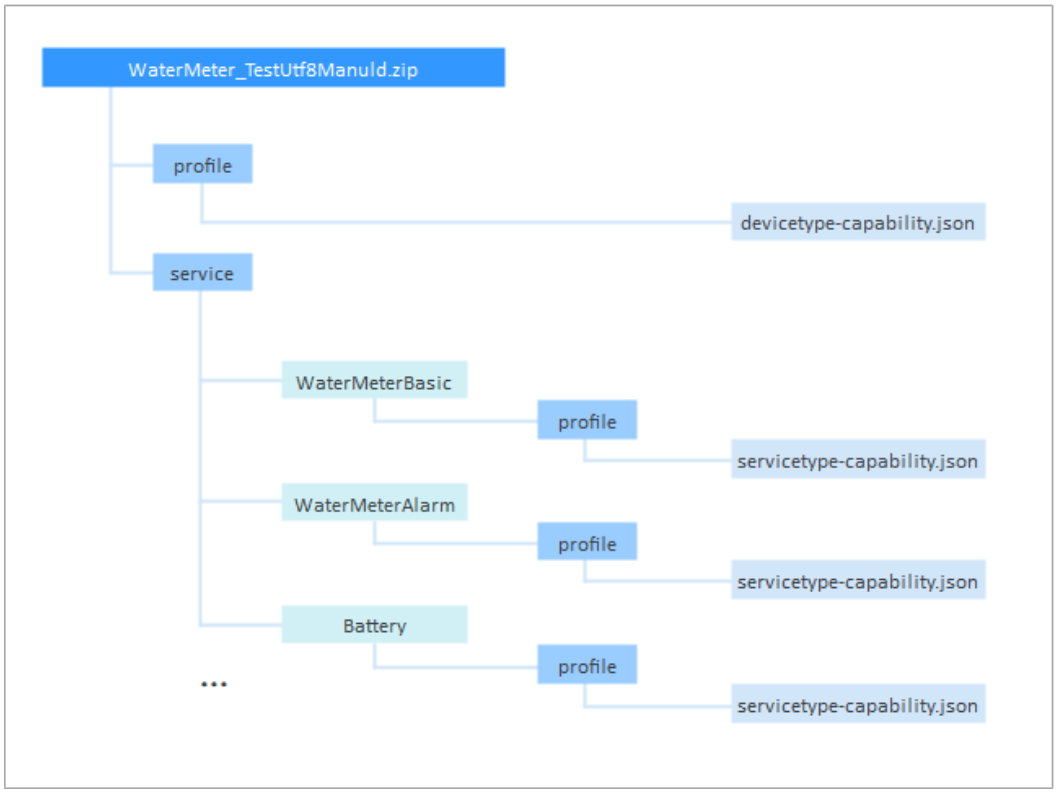


----End

3.3.3 Developing a Product Model Offline

Overview

A product model is essentially a ZIP package that combines one **devicetype-capability.json** file and several **serviceType-capability.json** files in the following hierarchy, in which **WaterMeter** indicates the device type, **TestUtf8Manuld** identifies the manufacturer ID, and **WaterMeterBasic**, **WaterMeterAlarm**, and **Battery** indicates the service types.



In this regard, defining an offline product model is to define device capabilities in the **devicetype-capability.json** file and service capabilities in the **servicetype-capability.json** files in JSON format based on the product model definition rules, which is time-consuming and requires familiarity with the JSON format.

Developing a Product Model Online is recommended.

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 in the Product Model	Value
Device Type	deviceType	WaterMeter
Manufacturer ID	manufacturerId	TestUtf8ManuId
Manufacturer Name	manufacturerName	HZYB
Protocol Type	protocolType	CoAP

Service list

Service	Service ID	Service Type	Value
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	Character string length. This field is valid only when **dataType** is set to **string**, **string list**, or **DateTime**.
-	-	-	unit	-	Yes	Unit. 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	List of enumerated values. 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.

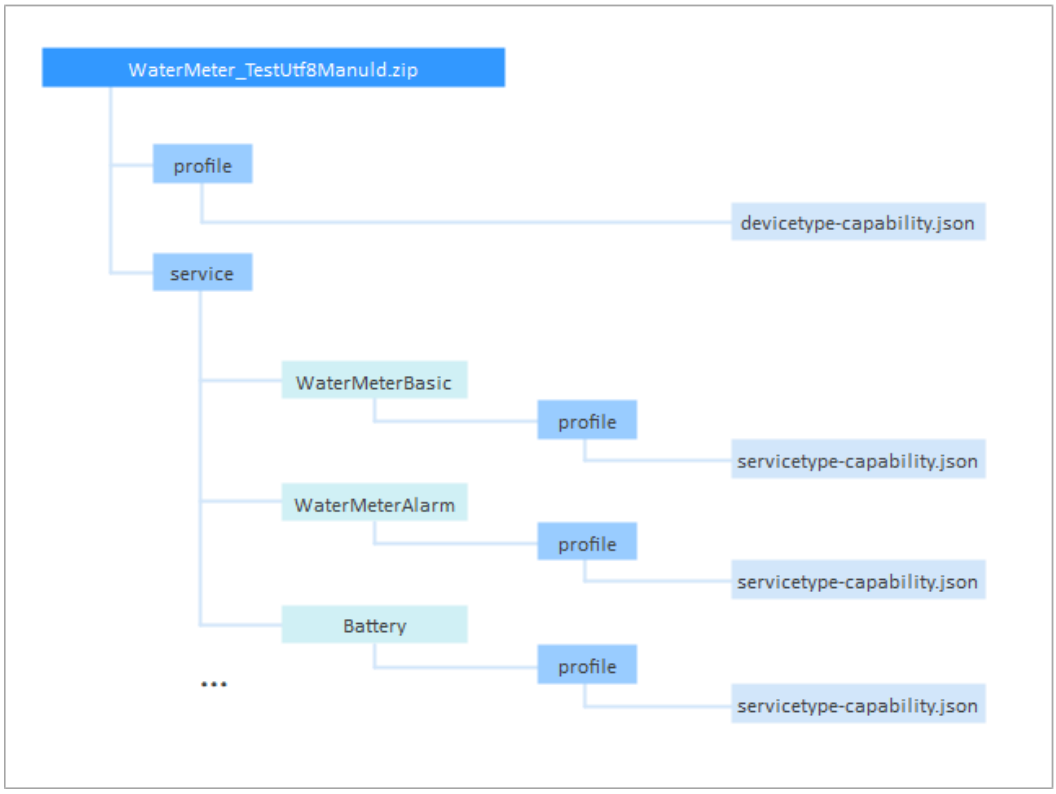
Field	Sub-field				Mandatory	Description
-	-	-	-	min	Yes	Minimum value. This field is valid only when **dataType** is set to **int** or **decimal**.
-	-	-	-	max	Yes	Maximum value. This field is valid only when **dataType** is set to **int** or **decimal**.
-	-	-	-	step	Yes	Step. This field is not used. Set it to **0**.
-	-	-	-	maxLength	Yes	Character string length. This field is valid only when **dataType** is set to **string**, **string list**, or **DateTime**.
-	-	-	-	unit	Yes	Unit. The value is determined by the parameter, for example: Temperature unit: C or K Percentage unit: % Pressure unit: Pa or kPa
-	-	-	-	enumList	Yes	List of enumerated values. 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	Name of a property.

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**.
-	-	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**.
-	-	method	-	-	Yes	Access mode. **R** indicates reading, **W** indicates writing, and **E** indicates subscription. Value: R, RW, RE, RWE, or null
-	-	unit	-	-	Yes	Unit. 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
-	-	maxLength	-	-	Yes	Character string length. This field is valid only when **dataType** is set to **string**, **string list**, or **DateTime**.
-	-	enumList	-	-	Yes	List of enumerated values. 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.

3.3.4 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.

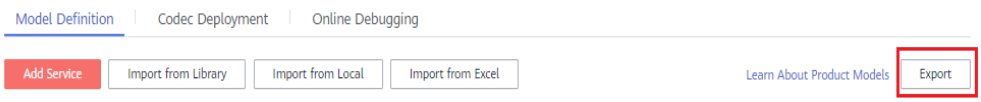
Exporting a Product Model

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**.

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.



----End

Importing a Product Model

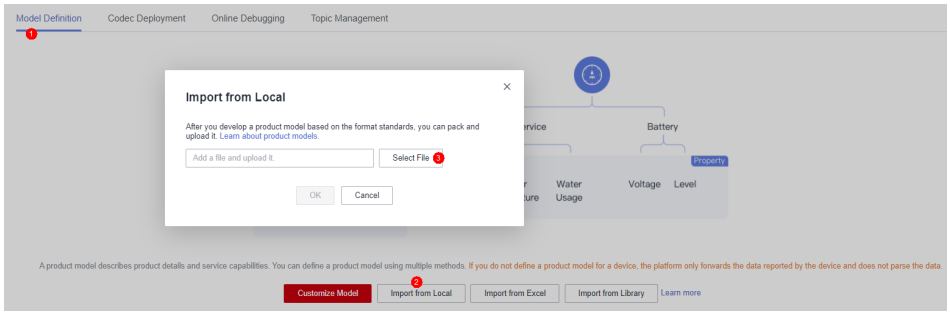
If you have a complete product model (developed offline or exported from other projects or platforms) or use an Excel file to develop a product model, you can import the product model to the platform.

NOTE

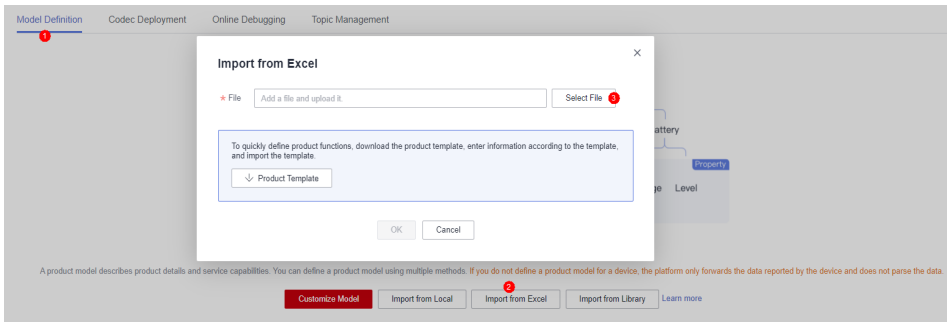
The product model imported from the local host does not contain a codec. If the device reports binary code, go to the IoTDA console to develop or import a codec.

- **Import from Local**
  a. Access the IoTDA service page and click **Access Console**.
  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 Local**. In the dialog box displayed, load the local product model and click **OK**.

Figure 3-7 Uploading a model file



- **Import from Excel**
  a. Access the IoTDA service page and click **Access Console**.
  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**.



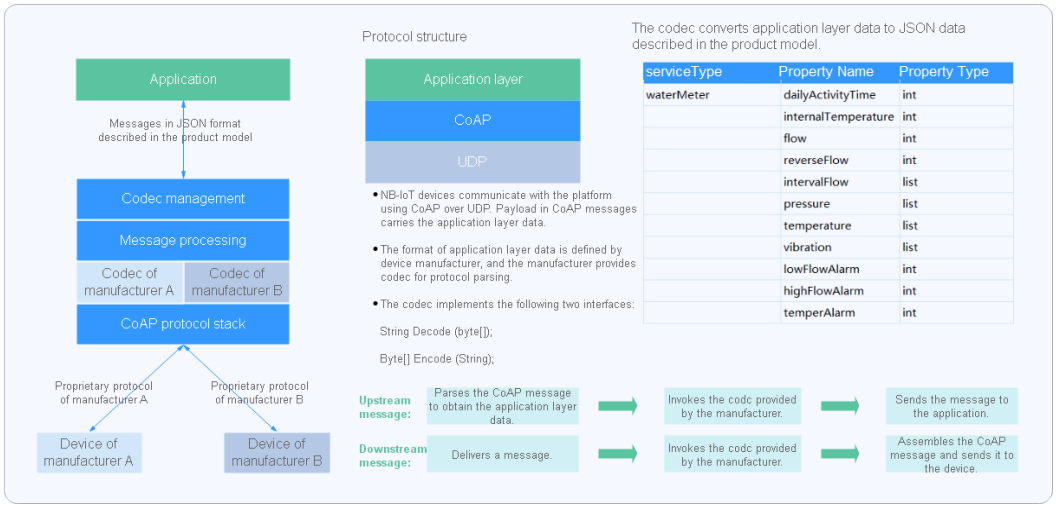
3.4 Developing a Codec

3.4.1 Codec Definition

IoTDA uses codecs to convert data between the binary and JSON formats as well as between JSON formats.

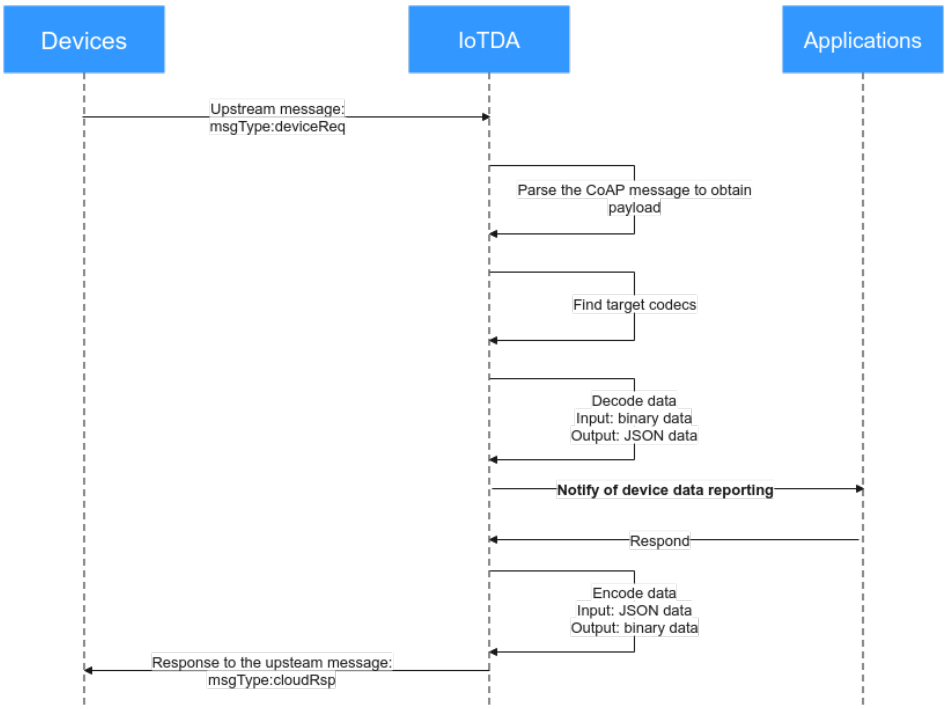
In the NB-IoT scenario, a codec can decode binary data reported by a device into the JSON format for the application to read, and encode the commands delivered

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.



Data Reporting

Figure 3-8 Codecs for data reporting

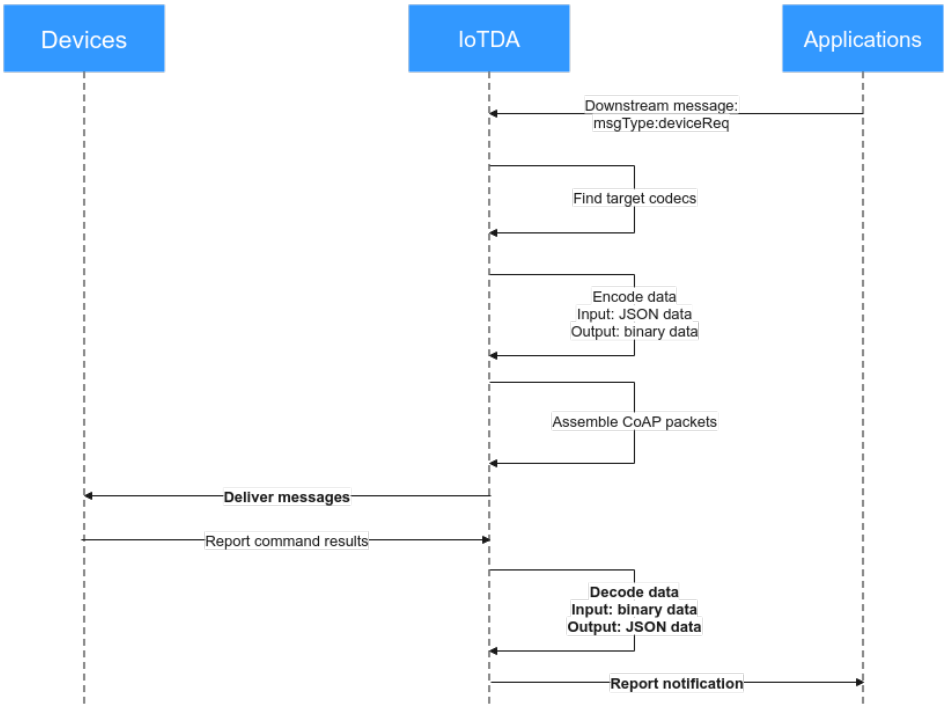


In the data reporting process, the codec is used in the following scenarios:

- 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 3-9 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 three 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.

3.4.2 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:

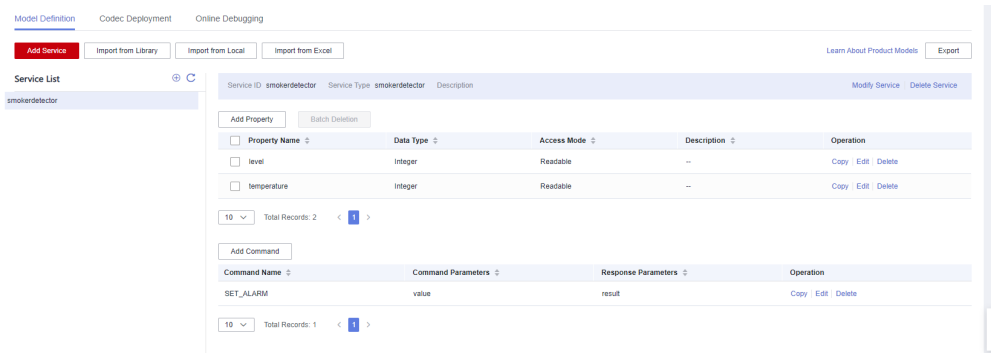
- Reporting smoke alarms (fire severity) and temperature
- Receiving and running remote control commands, which can be used to enable the alarm function remotely. For example, the smoke detector can report the temperature on the fire scene and remotely trigger a smoke alarm for evacuation.
- Reporting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

Defining a 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 the value **1** indicates that the alarm function is enabled.

Figure 3-10 Model definition - Smokerdetector



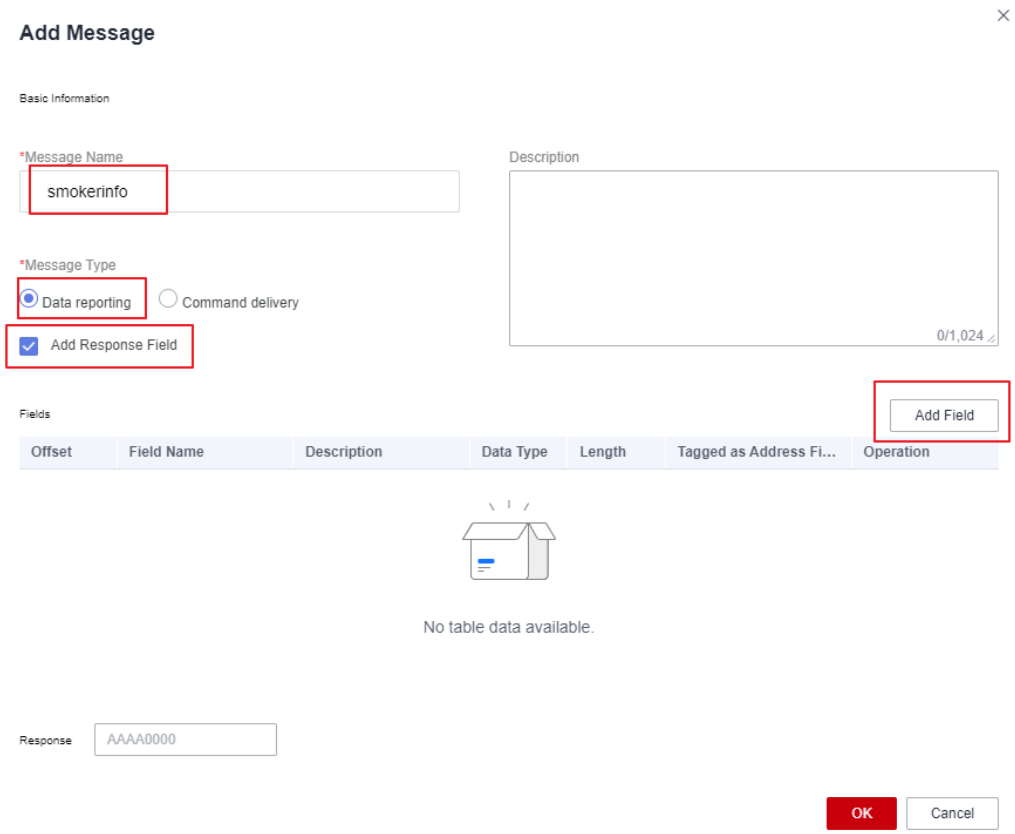
Developing a Codec

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)

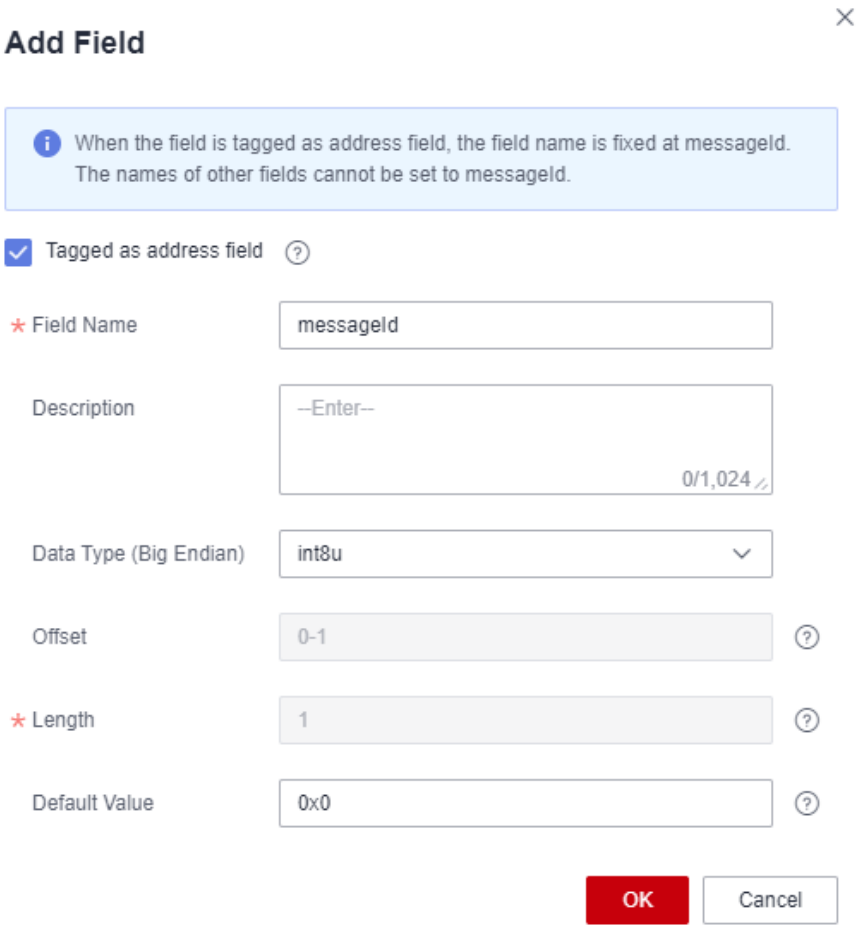
Figure 3-11 Adding a message - smokerinfo



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.

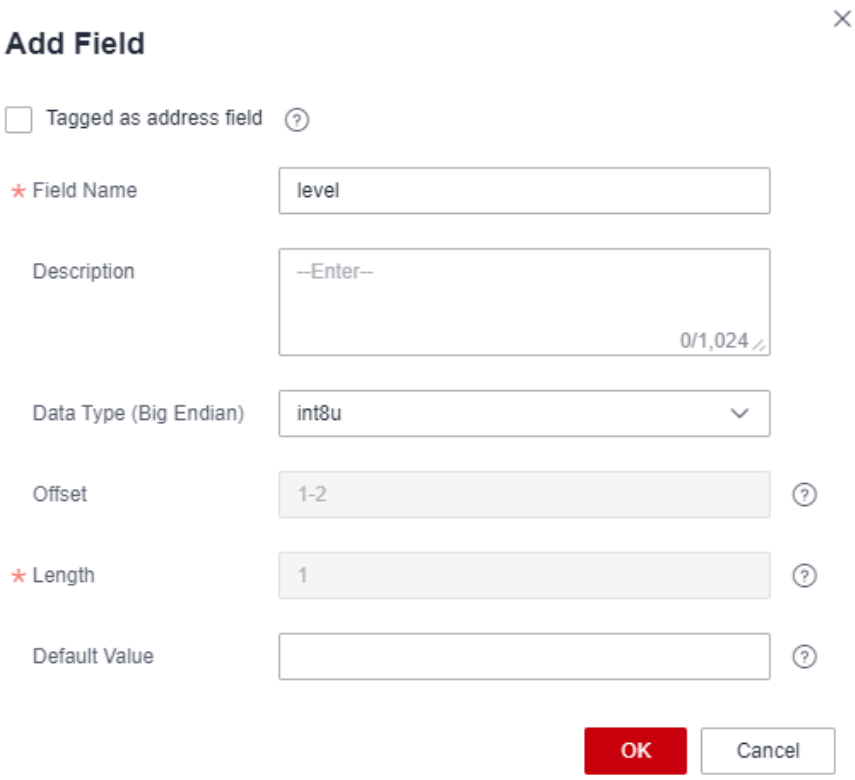
Figure 3-12 Adding a field - messageId



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**.

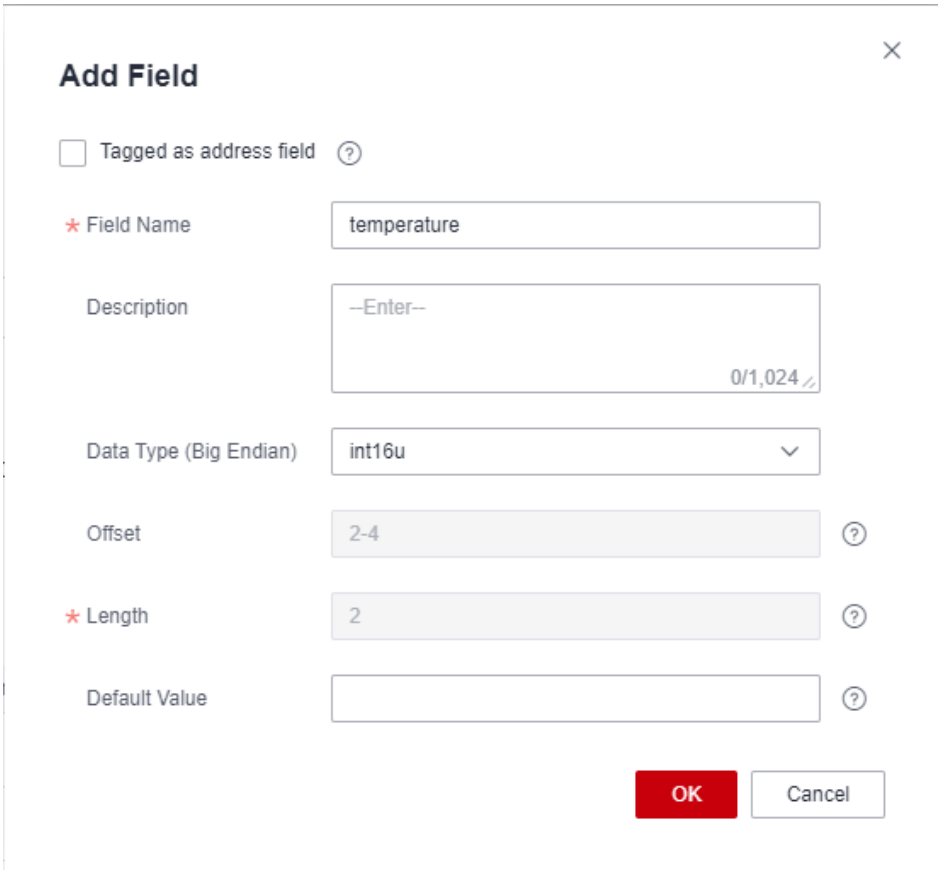
- The value of **Length** is automatically filled based on **Data Type**.
- **Default Value** can be left blank. If you do not set **Default Value**, the fire level is not fixed and has no default value.

Figure 3-13 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.

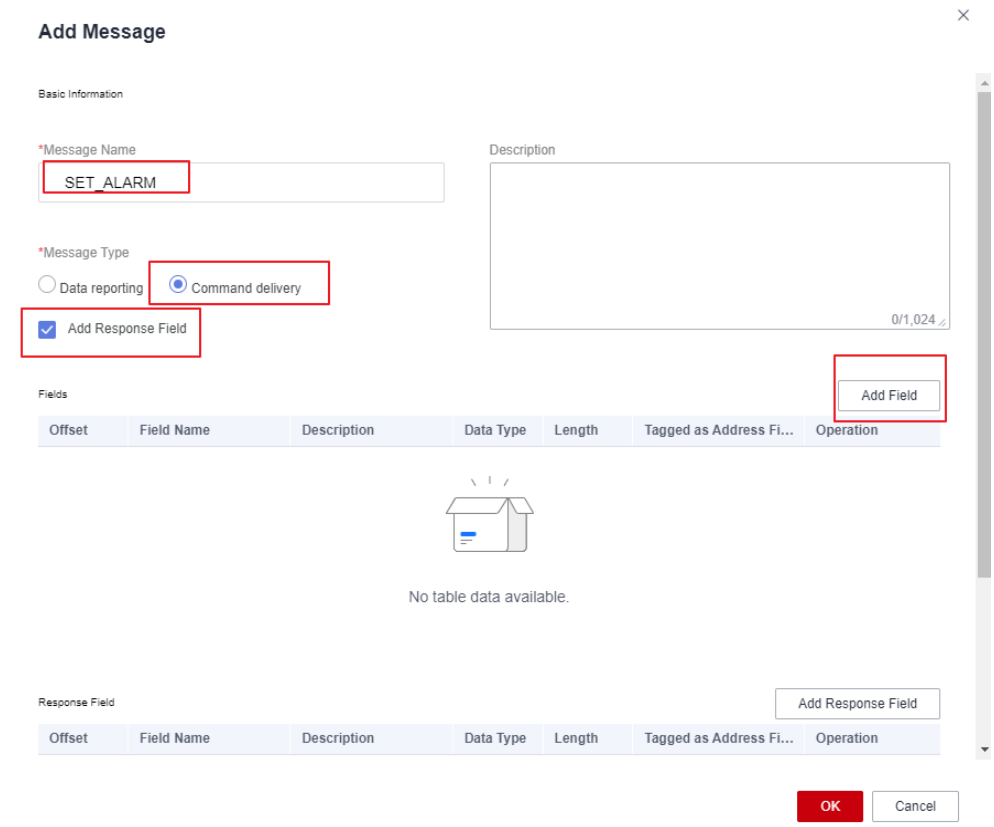
Figure 3-14 Adding a field - temperature



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:

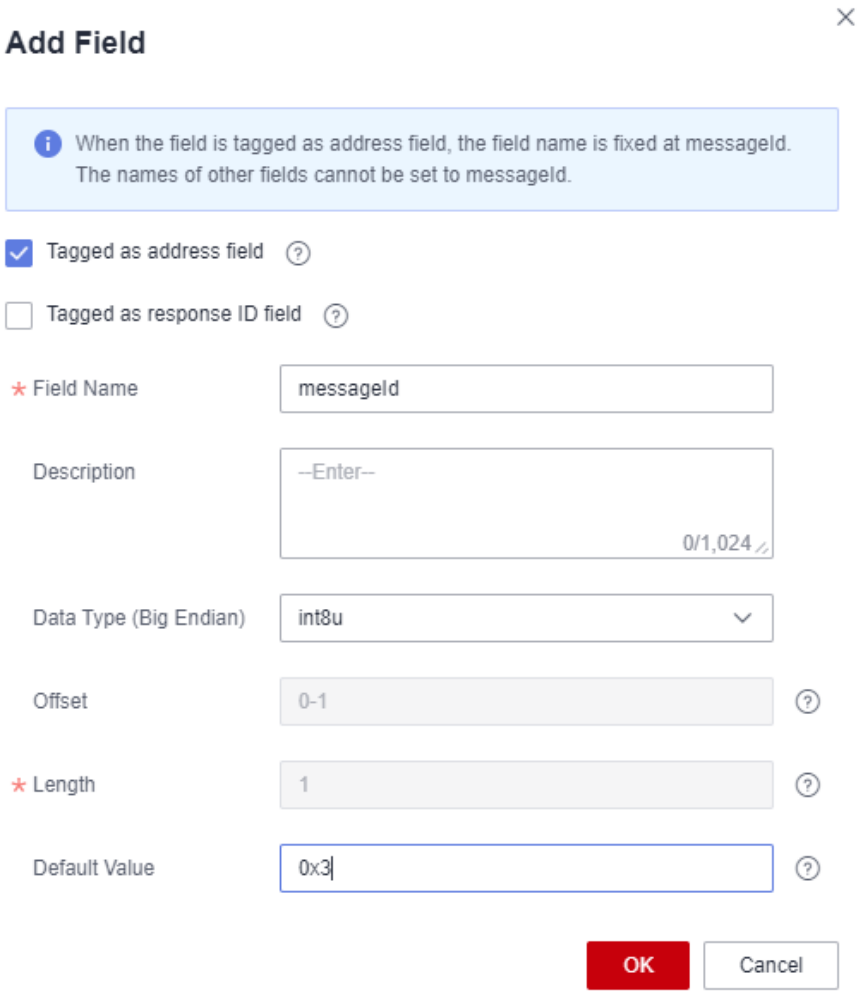
- Message Name: SET_ALARM
- Message Type: Command delivery
- Add Response Field: selected. After a response field is added, the device reports the command execution result after receiving the command. You can determine whether to add response fields as required.

Figure 3-15 Adding a message - SET_ALARM



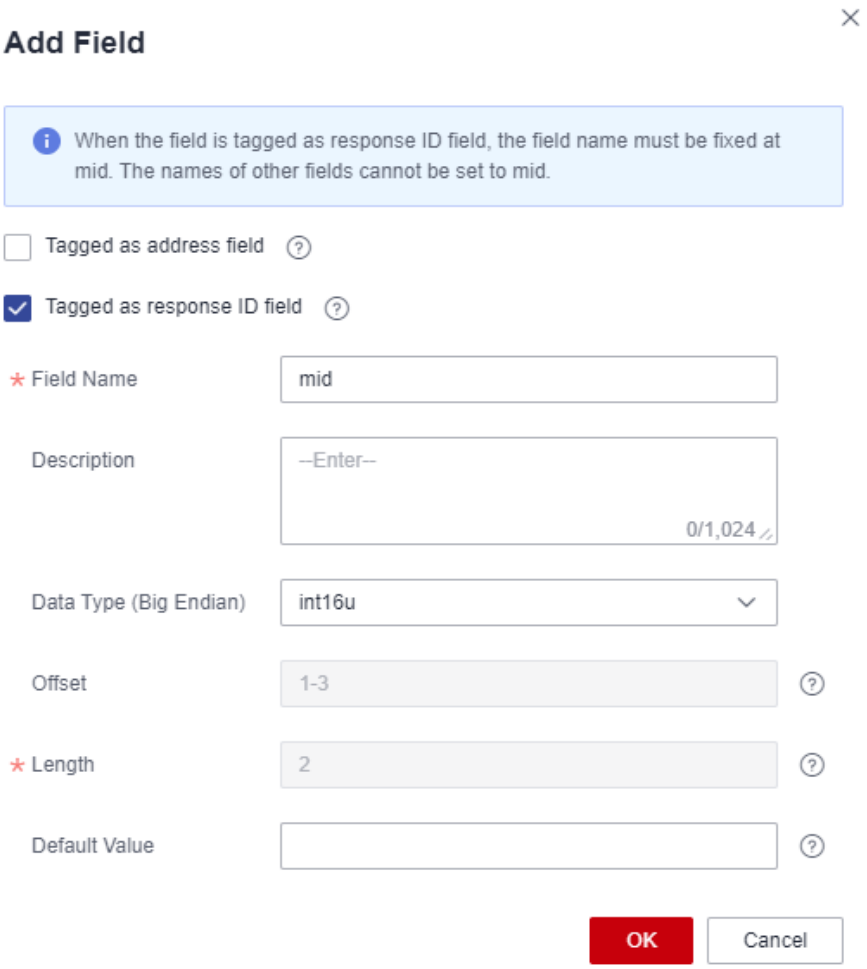
a. Click **Add Field** to add the **messageId** field, which indicates the message type. For example, set the message type of the fire alarm threshold to **0x3**. For details about the message ID, data type, length, default value, and offset, see **1**.

Figure 3-16 Adding a command field - messageId (0x3)



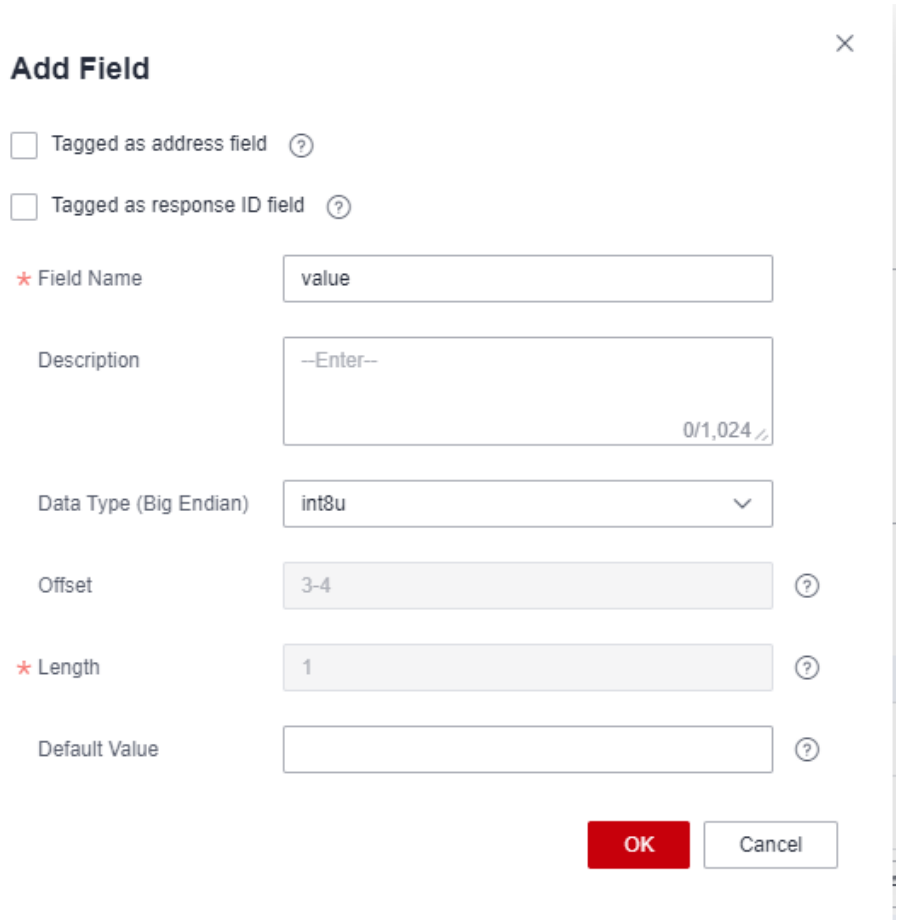
b. Add the **mid** field. This field is generated and delivered by the platform and is used to associate the delivered command with the command delivery response. The data type of the **mid** field is **int16u** by default. For details about the length, default value, and offset, see **2**.

Figure 3-17 Adding a command field - mid



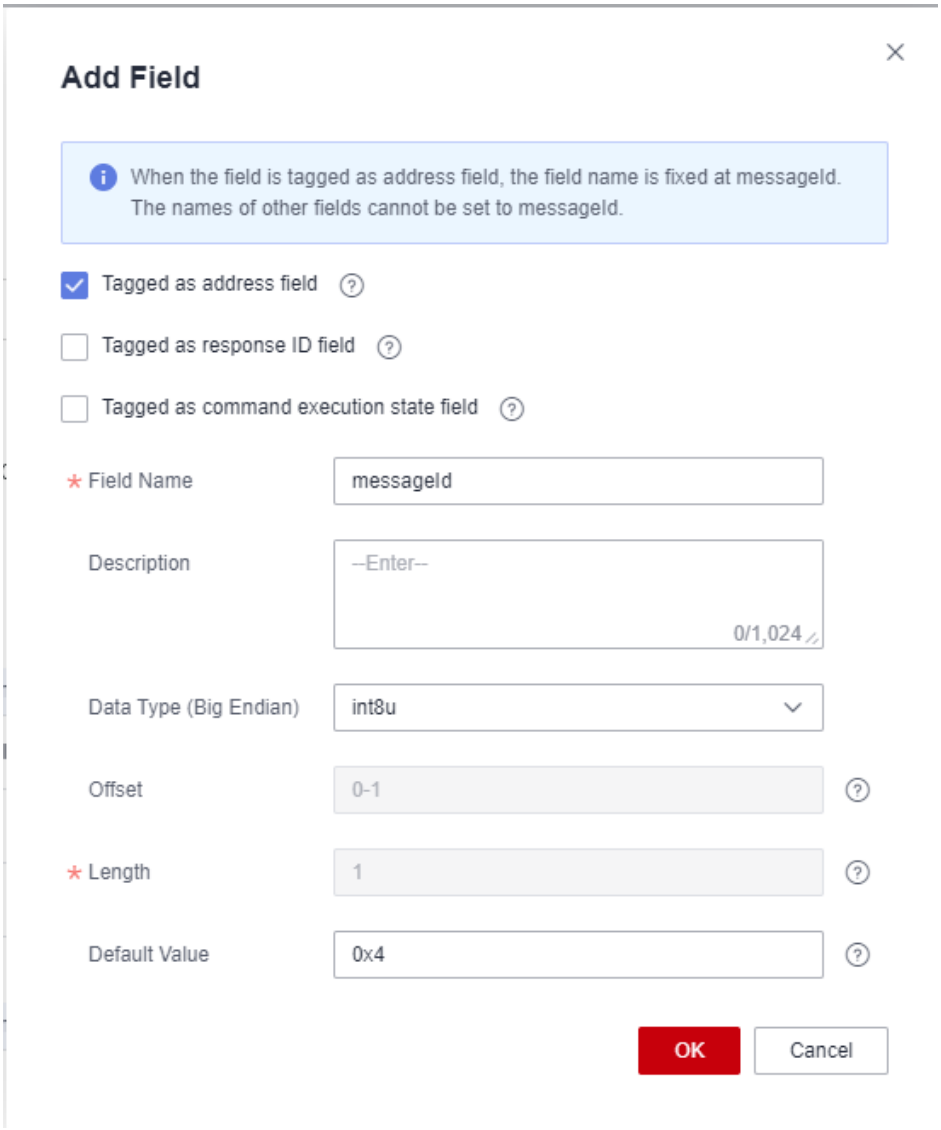
c. Add the **value** field to indicate the parameter value of the delivered command. For example, deliver the temperature threshold for a fire alarm. For details about the data type, length, default value, and offset, see 2.

Figure 3-18 Adding a command field - value



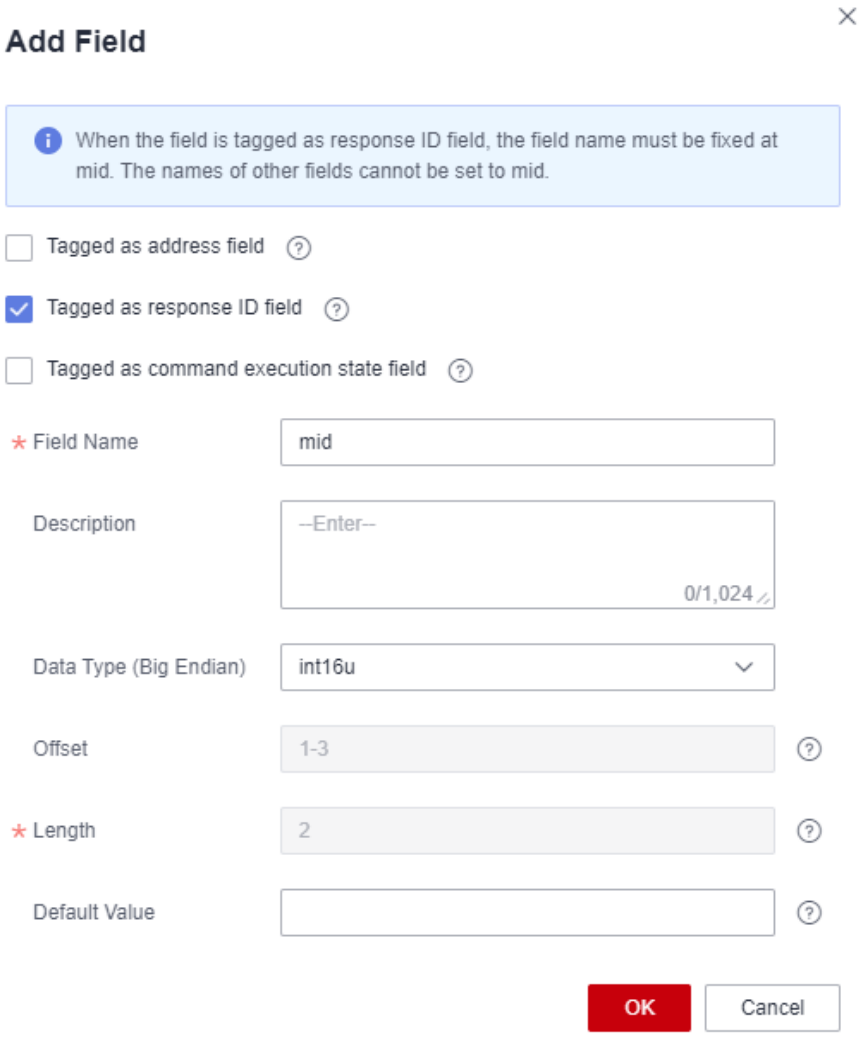
d. Click **Add Response Field** to add the **messageId** field, which indicates the message type. The command delivery response is an upstream message, which is differentiated from the data reporting message by the **messageId** field. The message type for reporting the temperature threshold of the fire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

Figure 3-19 Adding a response field - messageId (0x4)



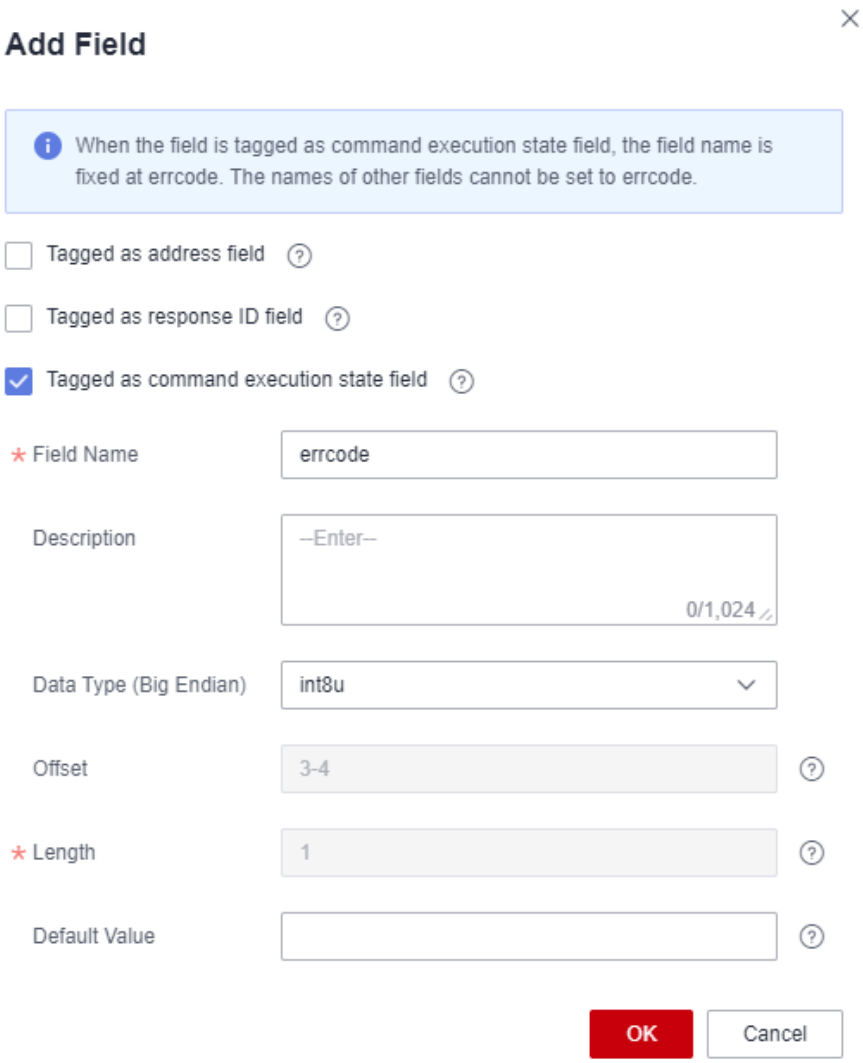
e. Add the **mid** field. This field must be the same as that in the command delivered by the IoT platform. It is used to associate the delivered command with the command execution result. The data type of the **mid** field is **int16u** by default. For details about the length, default value, and offset,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

Figure 3-20 Adding a response field - mid



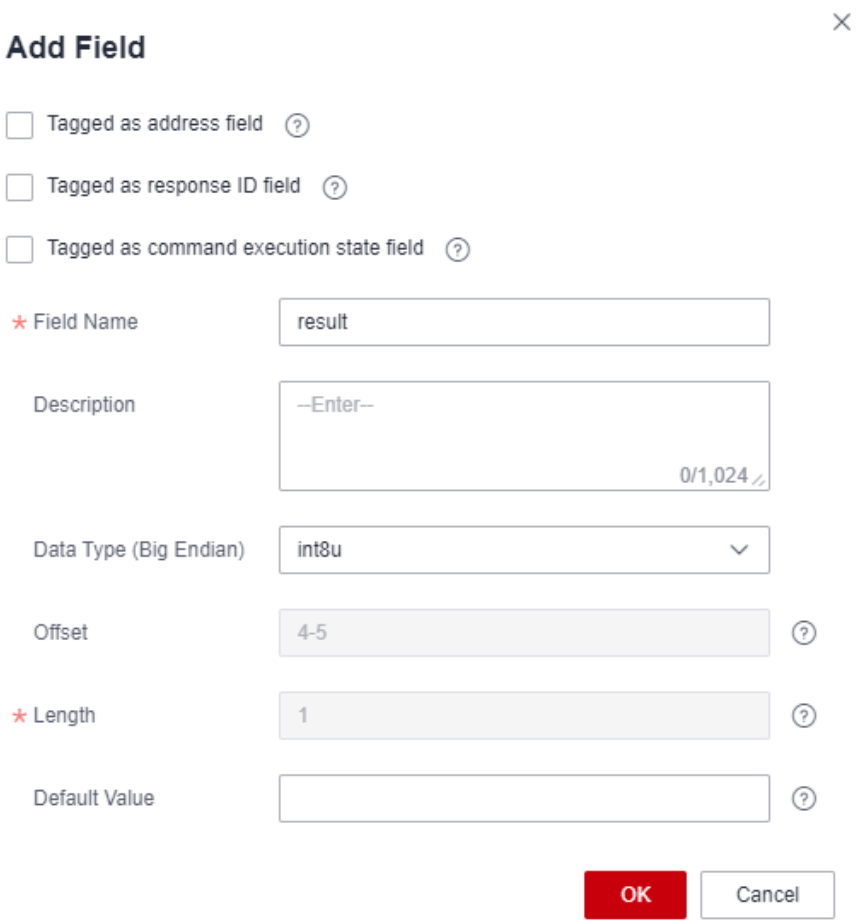
f. Add the **errcode** field to indicate the command execution status. **00** indicates success and **01** indicates failure. If this field is not carried in the response, the command is executed successfully by default. The data type of the **errcode** field is **int8u** by default. For details about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

Figure 3-21 Adding a response field - errcode



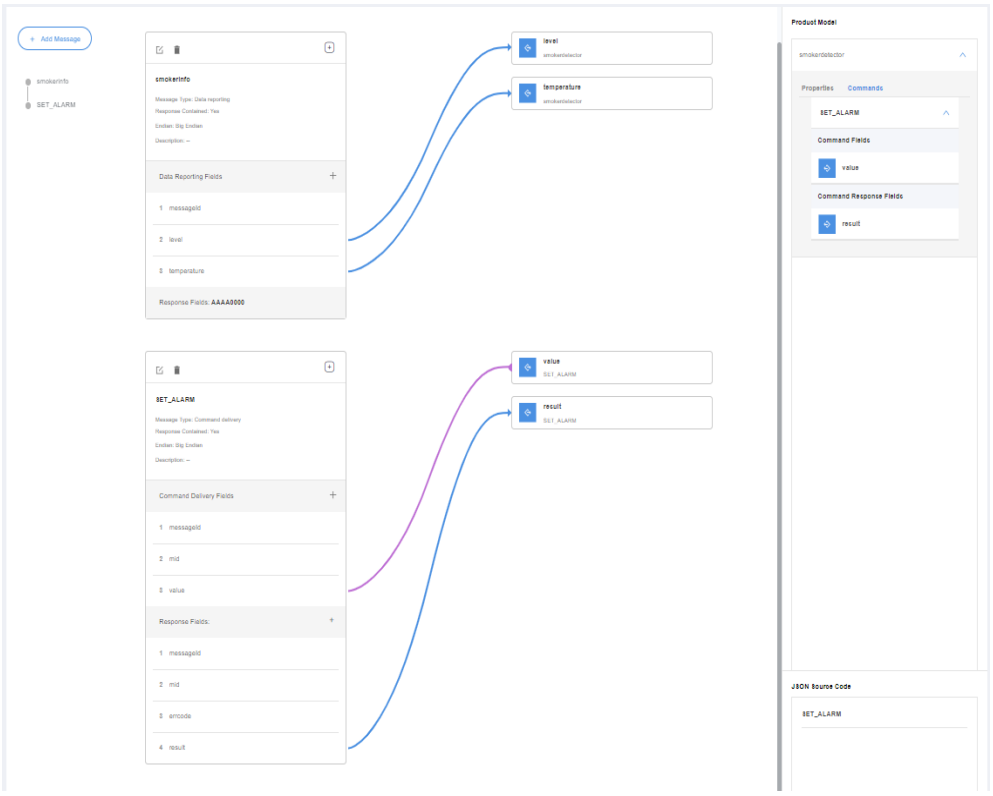
g. Add the **result** field to indicate the command execution result. For example, the device returns the current alarm threshold to the platform.

Figure 3-22 Adding a response field - result



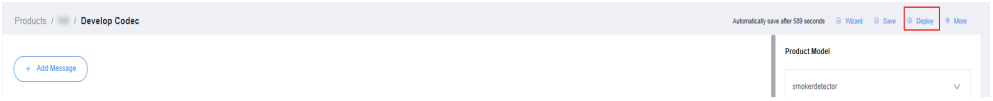
Step 4 Drag the property fields and command fields in **Device Model** on the right to set up a mapping between the fields in the data reporting message and those in the command delivery message.

Figure 3-23 Developing the smokerdetector codec online



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

Figure 3-24 Deploying a codec



----End

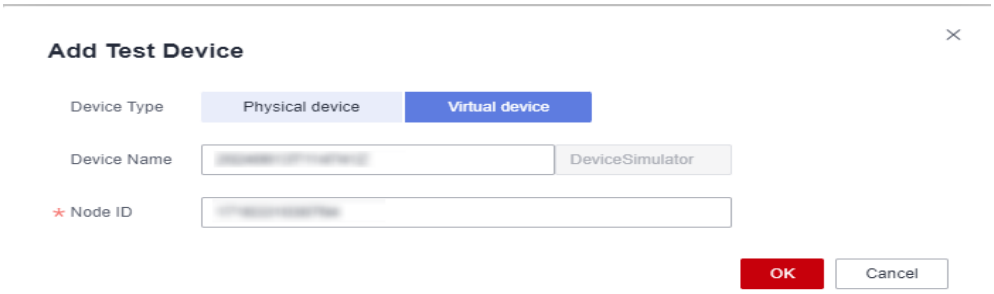
Testing the Codec

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 simulated device as an example to describe how to debug a codec.

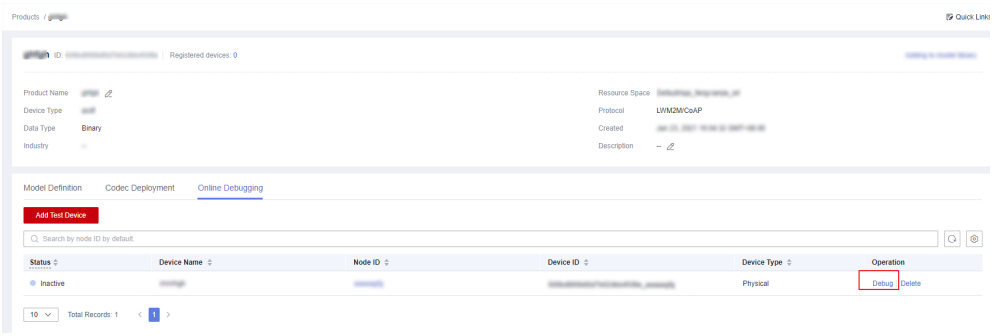
In the **Add Test Device** dialog box, select **Virtual device** for **Device Type** and click **OK**. The virtual device name contains **Simulator**. Only one virtual device can be created for each product.

Figure 3-25 Creating a virtual device



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

Figure 3-26 Entering debugging

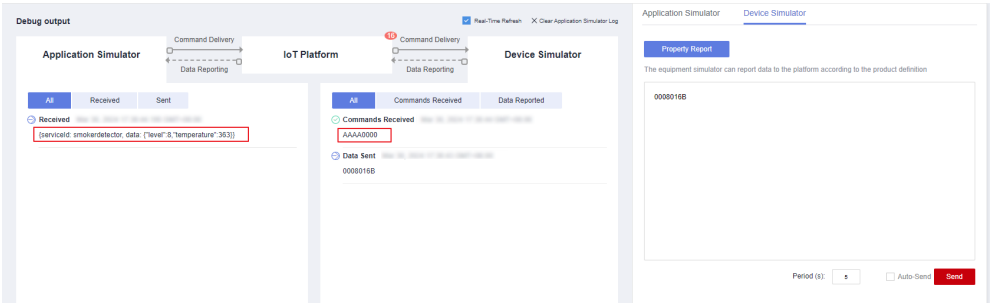


Step 4 Use the device simulator to report data. For example, a hexadecimal code stream (0008016B) is reported. **00** indicates the **messageId** field. **08** indicates the fire severity, and its length is one byte. **016B** indicates the temperature, and its length is two bytes.

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.

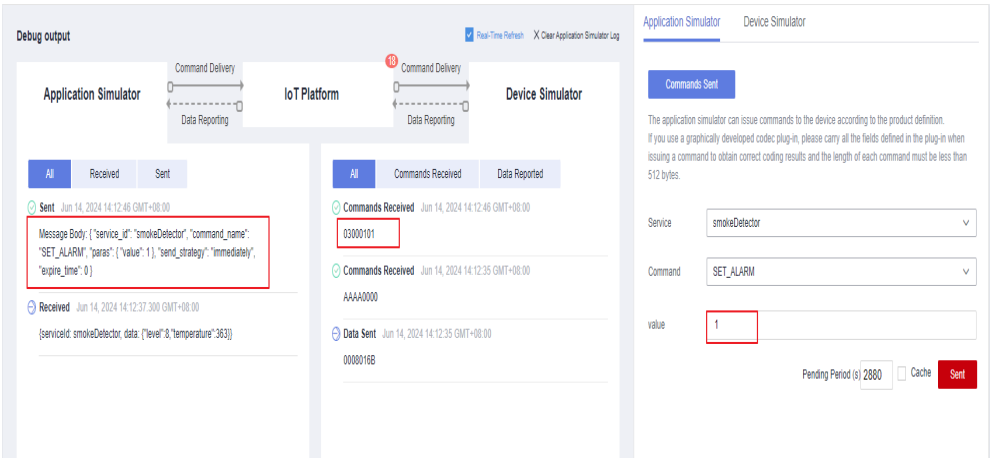
Figure 3-27 Simulating data reporting to smokerdetector



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 3-28 Simulating command delivery to 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.

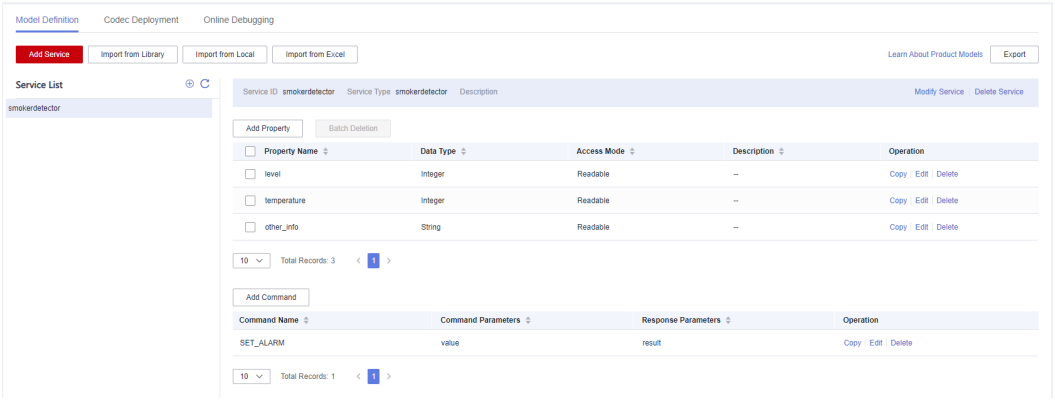
Codec for Strings and Variable-Length Strings

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

Defining a Product Model

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

Figure 3-29 Model definition - Smokerdetector carrying other_info



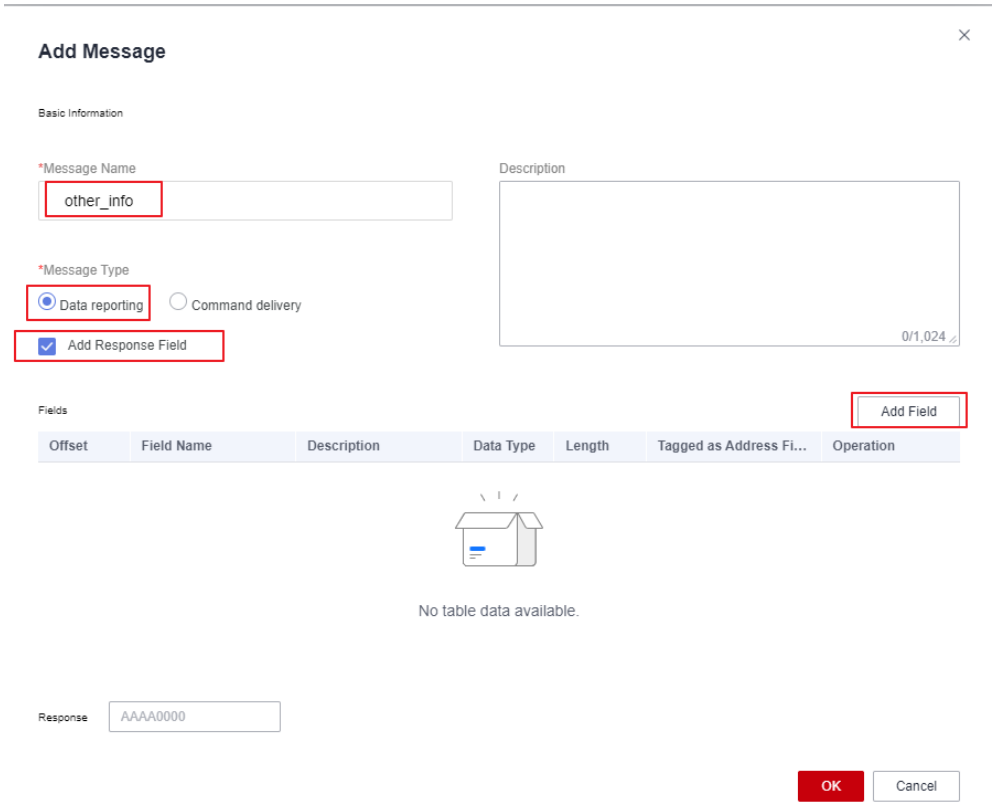
Developing a Codec

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:

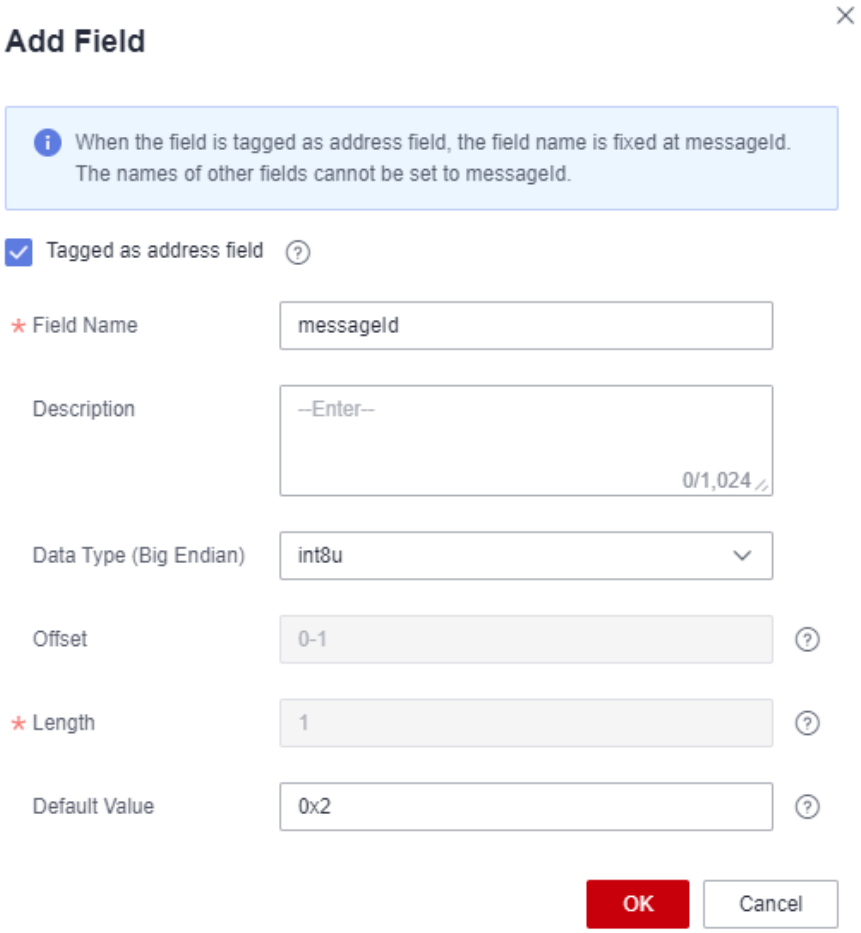
- **Message Name**: **other_info**
- **Message Type**: **Data reporting**
- **Add Response Field**: selected After response fields are added, the platform delivers the response data set by the application to the device after receiving the data reported by the device.
- **Response**: AAAA0000 (default)

Figure 3-30 Adding a message - other_info



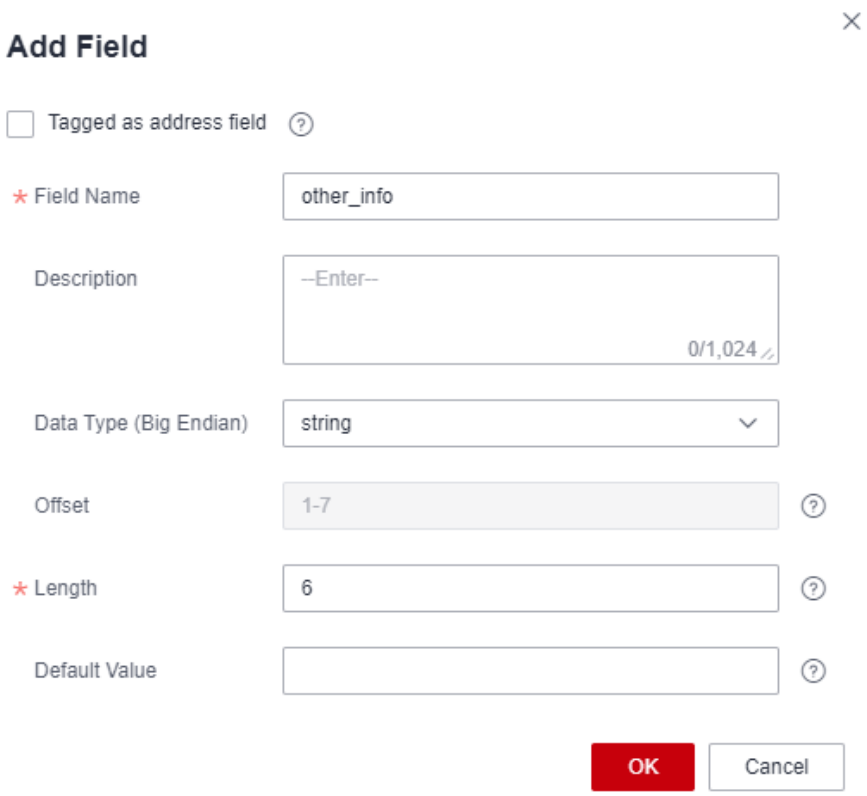
1. Click **Add Field** to add the **messageId** field, which indicates the message type. In this scenario, the value **0x0** is used to identify the message that reports the fire severity and temperature, **0x1** is used to identify the message that reports only the temperature, and **0x2** is used to 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 3-31 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**.

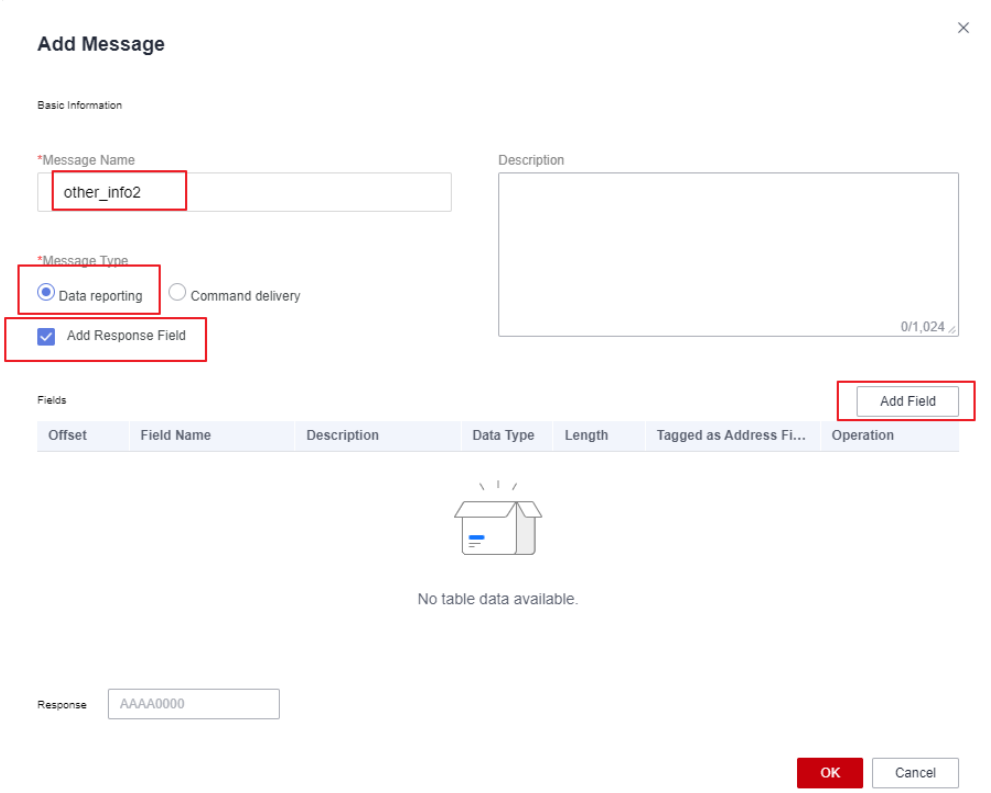
Figure 3-32 Adding a field - other_info



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:

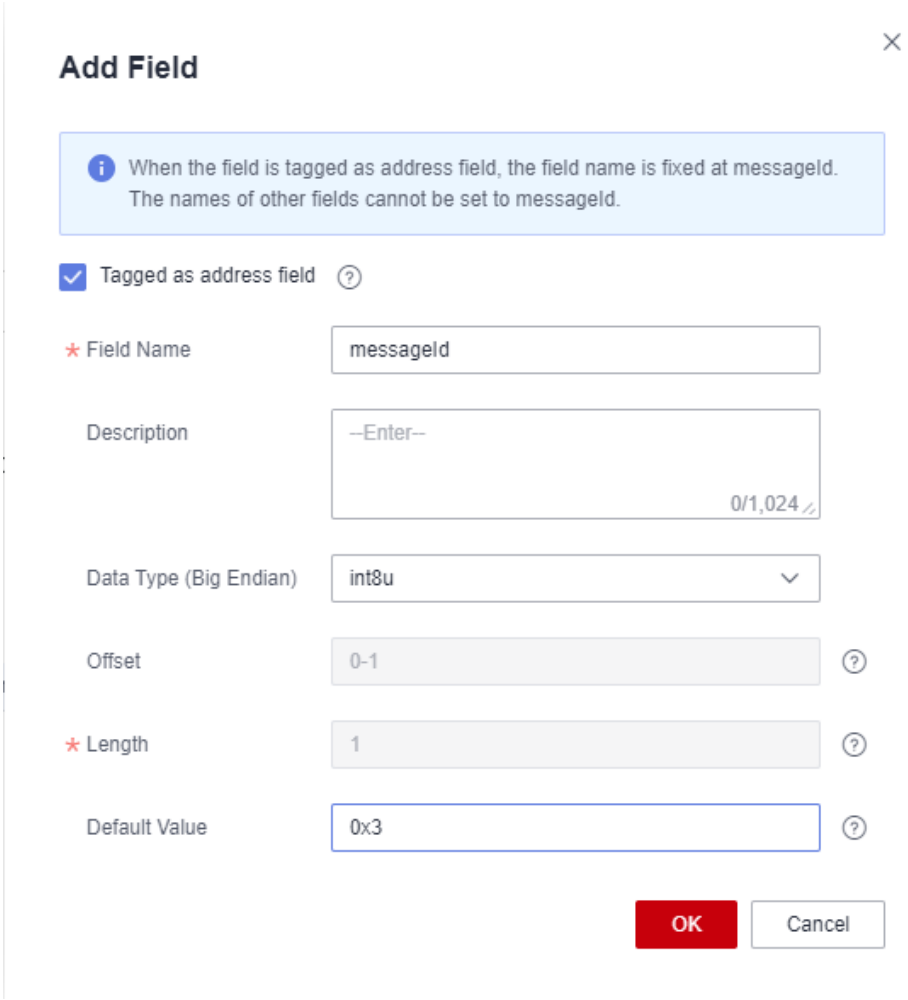
- **Message Name**: other_info2
- **Message Type**: Data reporting
- **Add Response Field**: selected. After response fields are added, the platform delivers the response data set by the application to the device after receiving the data reported by the device.
- **Response**: AAAA0000 (default)

Figure 3-33 Adding a message - other_info2



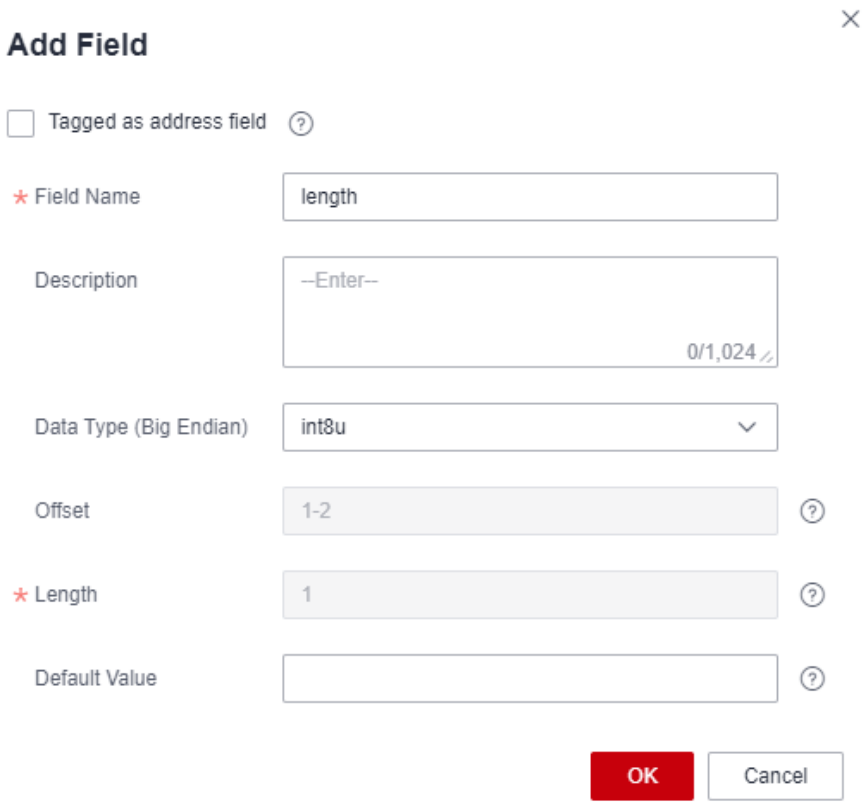
1. Add the **messageId** field to indicate the message type. In this scenario, the value **0x0** is used to identify the message that reports the fire severity and temperature, **0x1** is used to identify the message that reports only the temperature, and **0x3** is used to identify the message that 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 3-34 Adding a field - messageId (0x3)



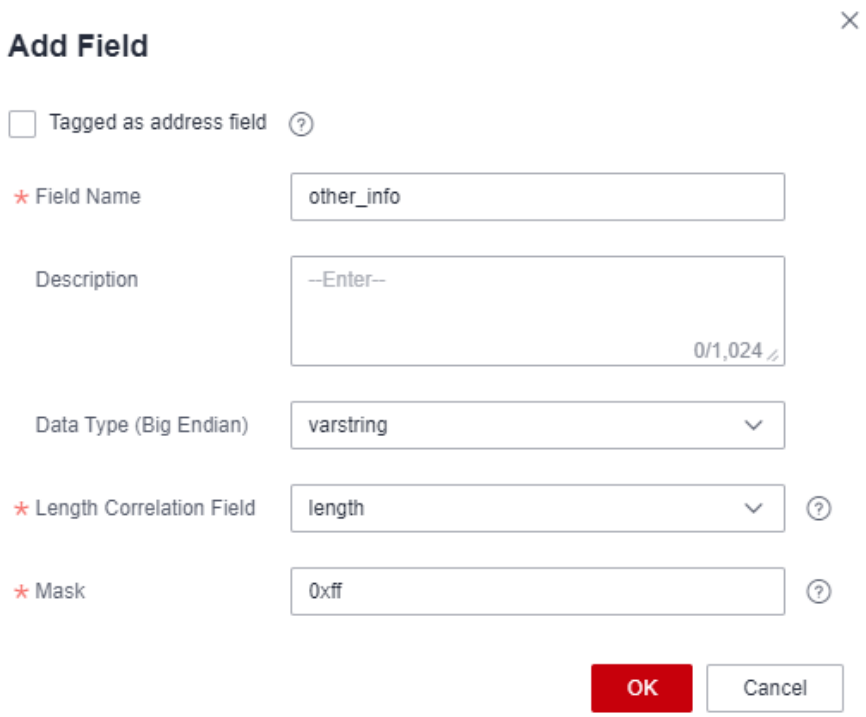
2. Add the **length** field to indicate the length of a variable-length string. **Data Type** is configured based on the length of the variable-length string. If the string contains 255 or fewer characters in this scenario, set this parameter to **int8u**. For details about the length, default value,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

Figure 3-35 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**.

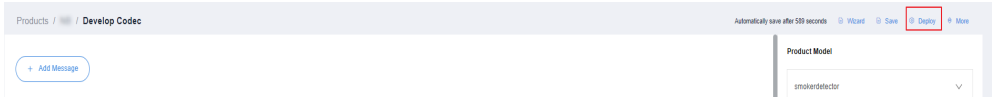
Figure 3-36 Adding a field - other_info as varstring



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.

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

Figure 3-37 Deploying a codec



----End

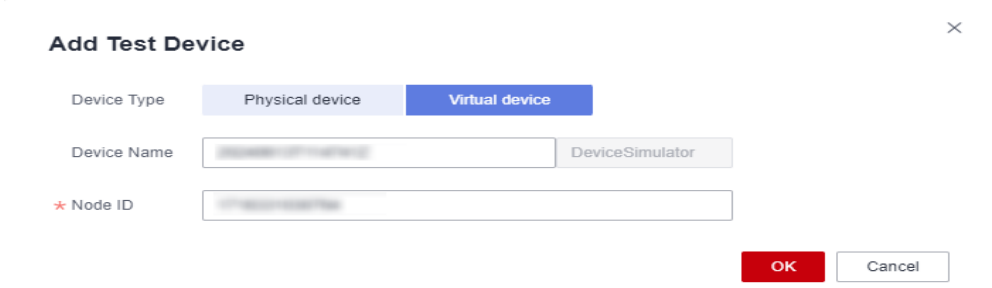
Testing the Codec

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 simulated device as an example to describe how to debug a codec.

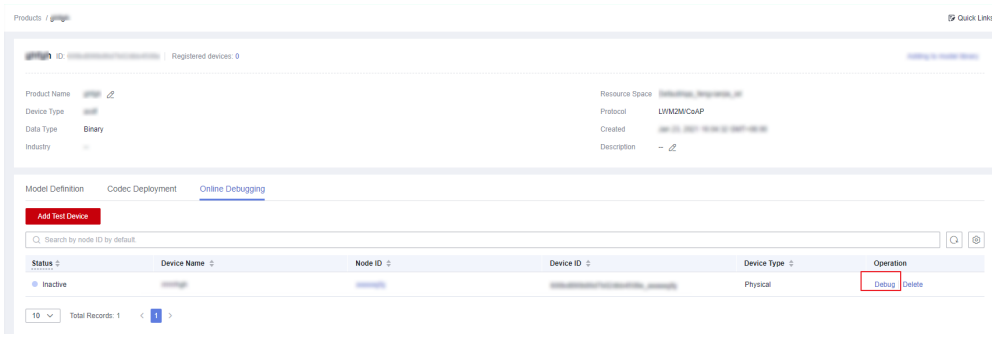
In the **Add Test Device** dialog box, select **Virtual device** for **Device Type** and click **OK**. The virtual device name contains **Simulator**. Only one virtual device can be created for each product.

Figure 3-38 Creating a virtual device



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

Figure 3-39 Entering debugging

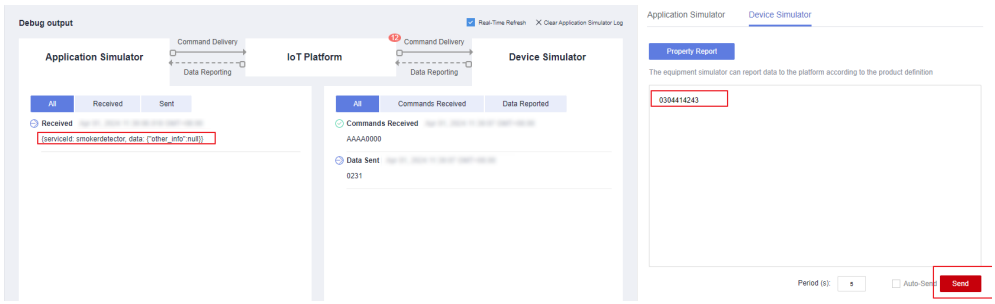


Step 4 Use the device simulator to report the description of the string type.

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.

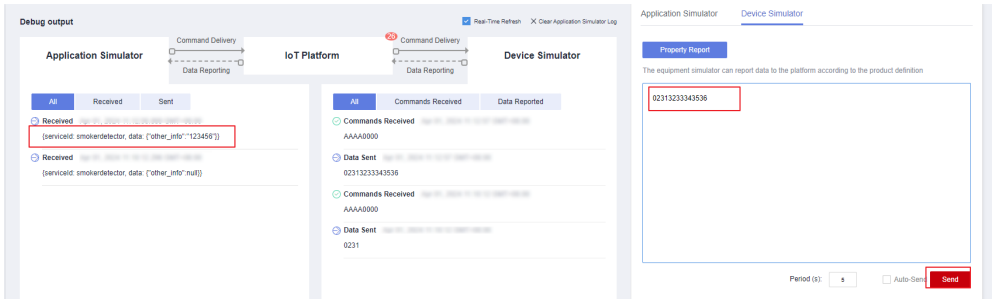
Figure 3-40 Simulating data reporting - other_info too short



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.

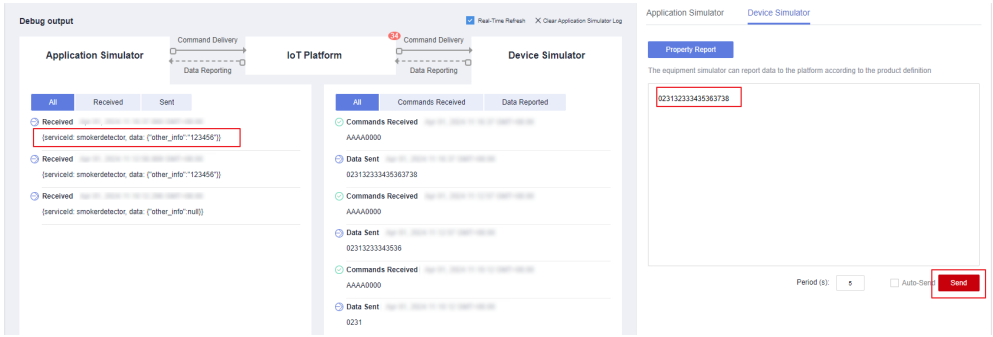
Figure 3-41 Simulating data reporting - other_info length proper



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.

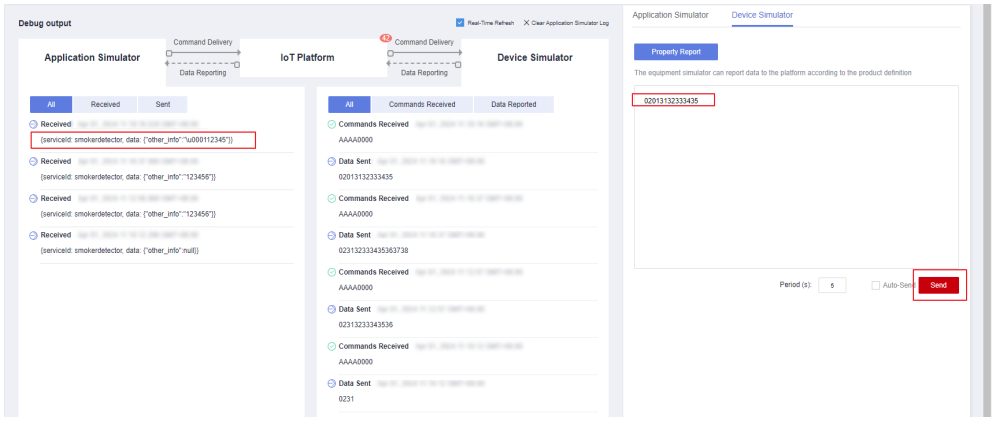
Figure 3-42 Simulating data reporting - other_info too long



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 3-43 Simulating data reporting - other_info as ASCII code

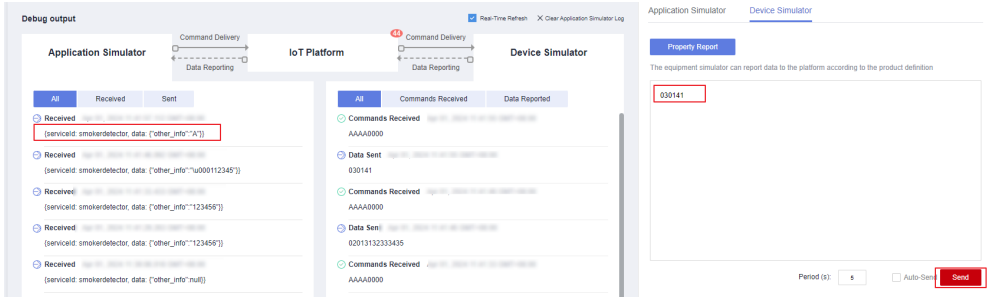


Step 5 Use the device simulator to report the description of the variable-length string type.

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.

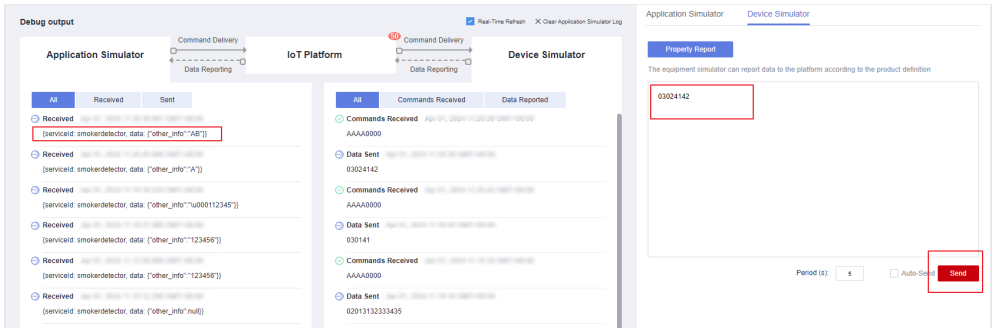
Figure 3-44 Simulating data reporting - other_info as variable-length character string 1



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.

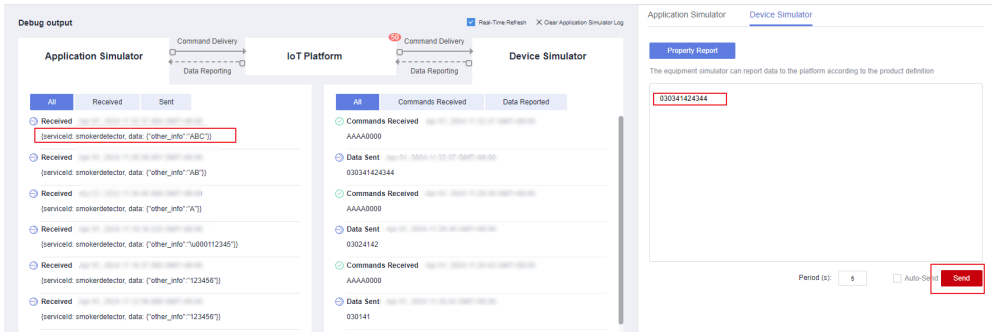
Figure 3-45 Simulating data reporting - other_info as variable-length character string 2



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.

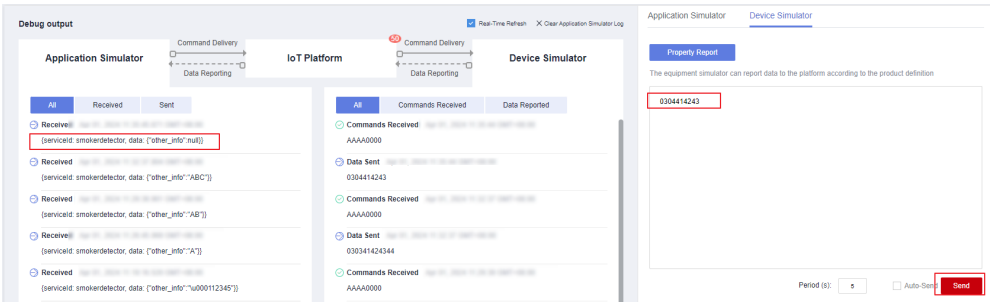
Figure 3-46 Simulating data reporting - other_info as variable-length character string 3



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 3-47 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.

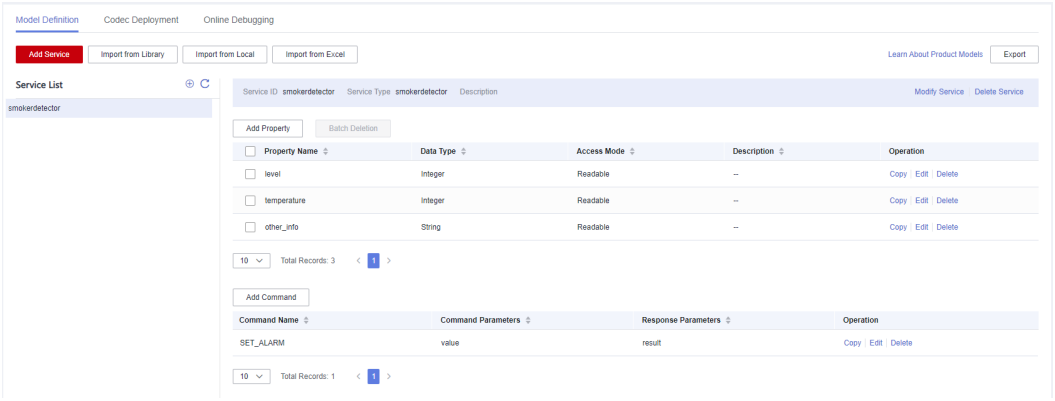
Codec for Arrays and Variable-Length Arrays

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

Defining a Product Model

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

Figure 3-48 Model definition - Smokerdetector carrying other_info



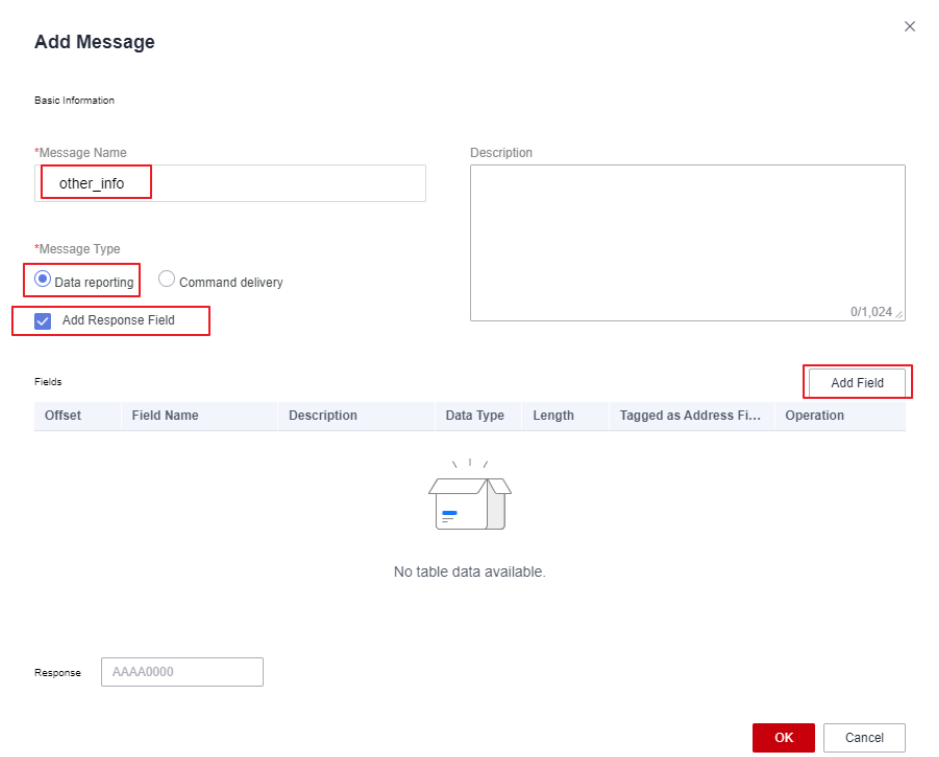
Developing a Codec

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:

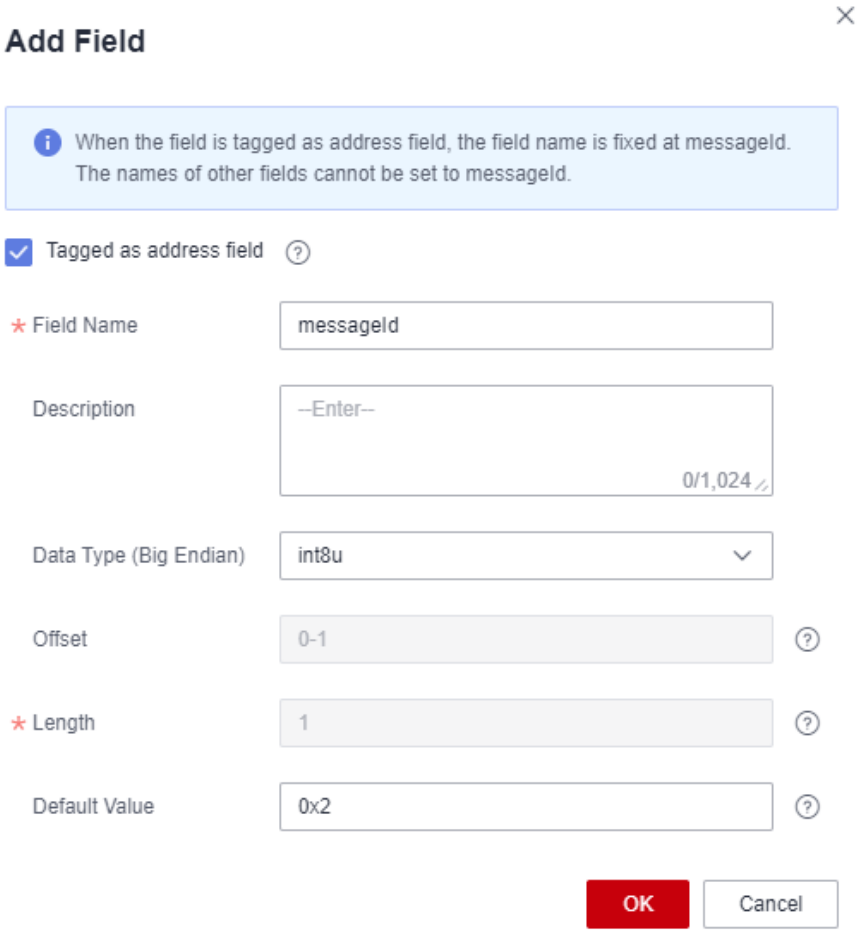
- **Message Name**: **other_info**
- **Message Type**: **Data reporting**
- **Add Response Field**: selected After response fields are added, the platform delivers the response data set by the application to the device after receiving the data reported by the device.
- Response: AAAA0000 (default)

Figure 3-49 Adding a message - other_info



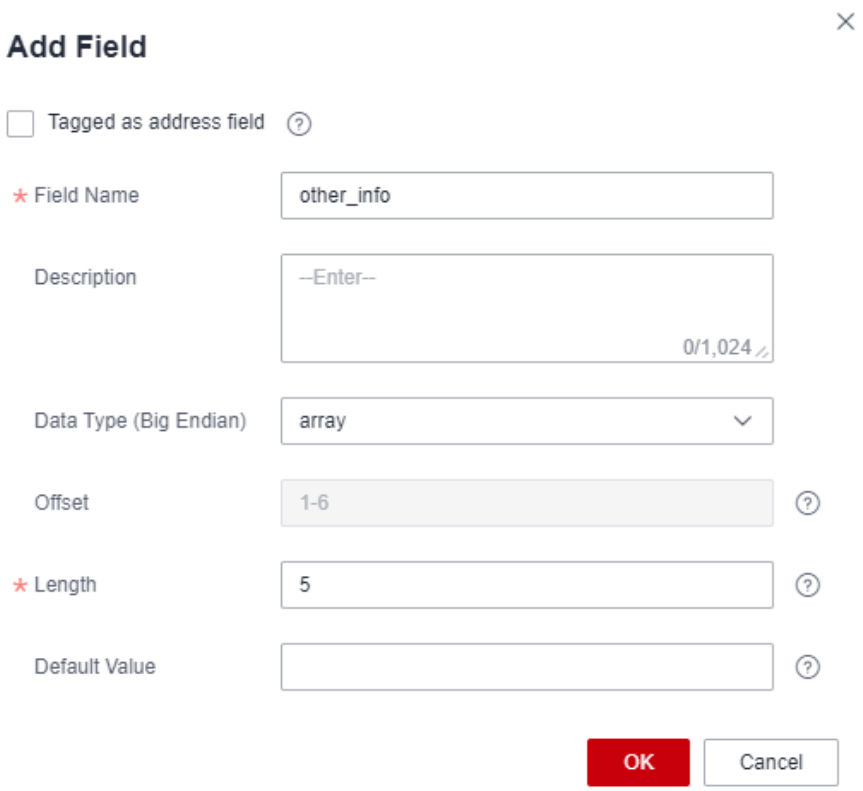
1. Click **Add Field** to add the **messageId** field, which indicates the message type. In this scenario, the value **0x0** is used to identify the message that reports the fire severity and temperature, **0x1** is used to identify the message that reports only the temperature, and **0x2** is used to 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 3-50 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.

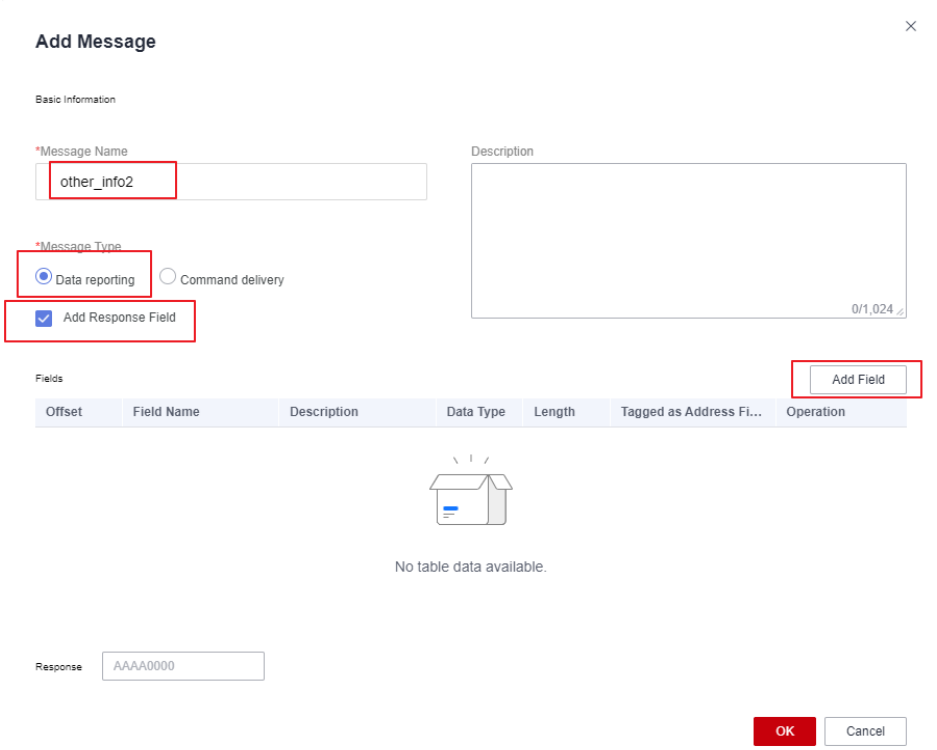
Figure 3-51 Adding a field - other_info as array



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:

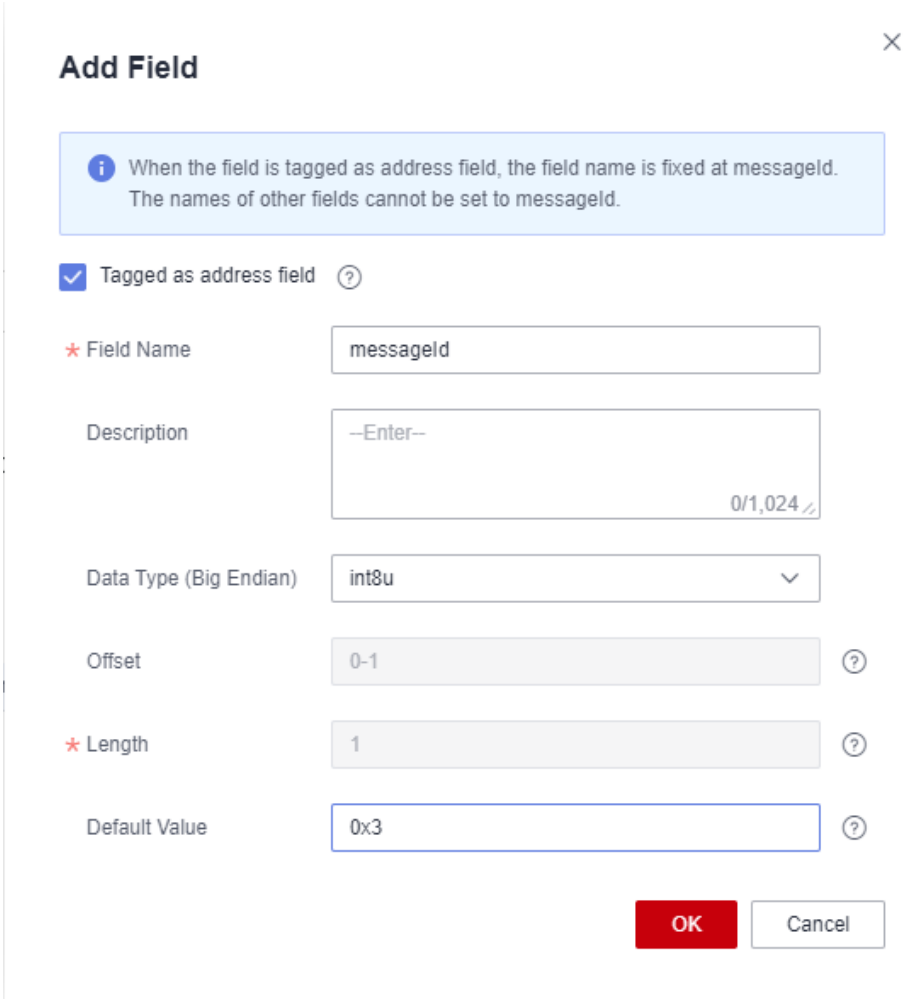
- **Message Name**: other_info2
- **Message Type**: Data reporting
- **Add Response Field**: selected. After response fields are added, the platform delivers the response data set by the application to the device after receiving the data reported by the device.
- **Response**: AAAA0000 (default)

Figure 3-52 Adding a message - other_info2



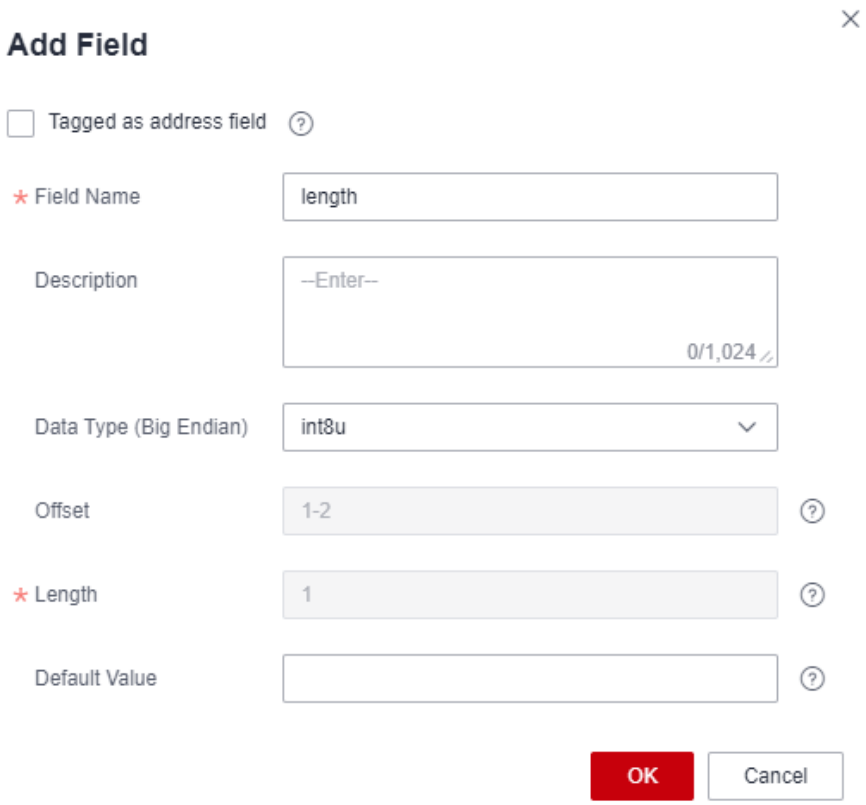
1. Click **Add Field** to add the **messageId** field, which indicates the message type. In this scenario, the value **0x0** is used to identify the message that reports the fire severity and temperature, **0x1** is used to identify the message that reports only the temperature, and **0x3** is used to 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 3-53 Adding a field - messageId (0x3)



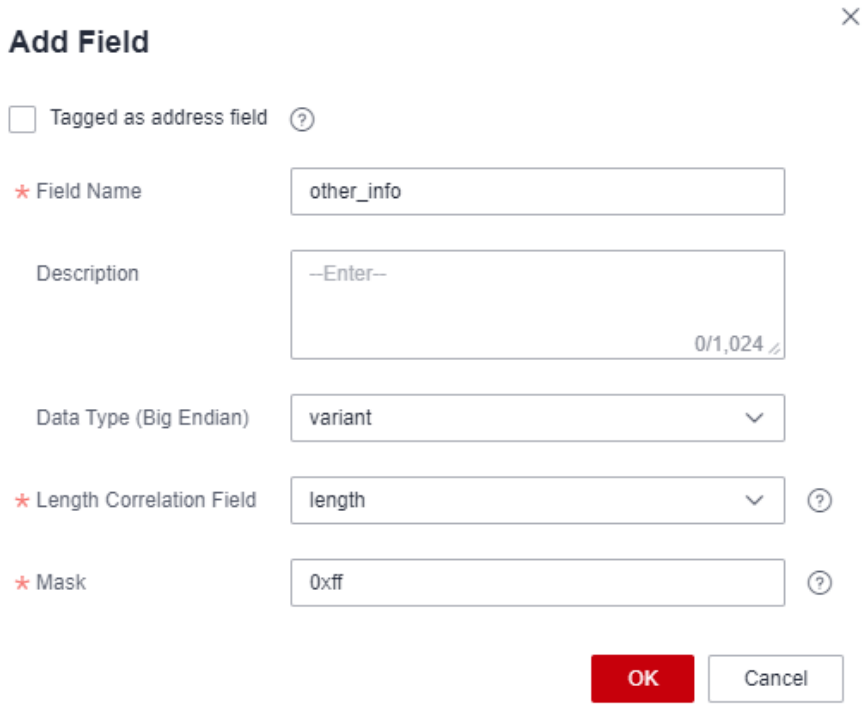
2. Add the **length** field to indicate the length of an array. **Data Type** is configured based on the length of the variable-length array. If the array contains 255 or fewer characters, set this parameter to **int8u**. For details about the length, default value, and offset, see **2**.

Figure 3-54 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**.

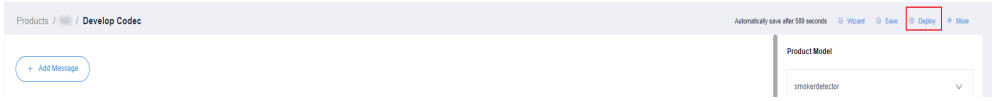
Figure 3-55 Adding a field - other_info as variant



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.

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

Figure 3-56 Deploying a codec



----End

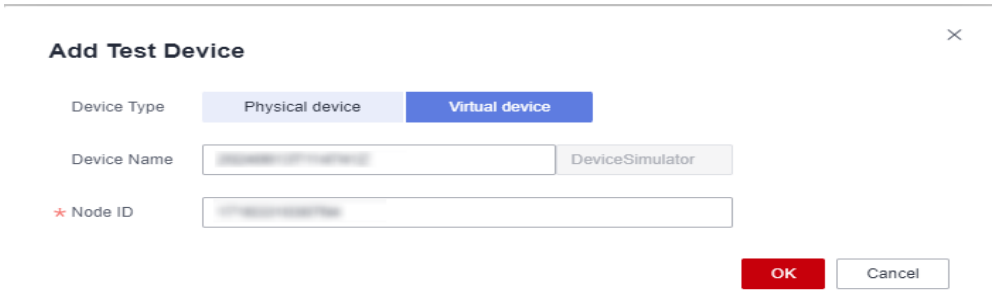
Testing the Codec

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 simulated device as an example to describe how to debug a codec.

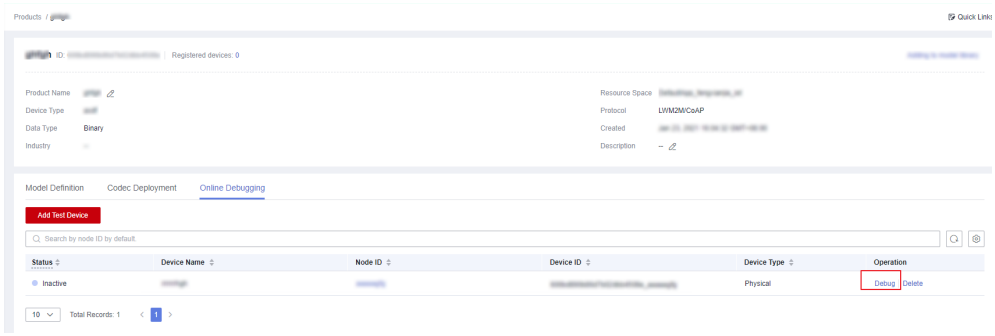
In the **Add Test Device** dialog box, select **Virtual device** for **Device Type** and click **OK**. The virtual device name contains **Simulator**. Only one virtual device can be created for each product.

Figure 3-57 Creating a virtual device



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

Figure 3-58 Entering debugging

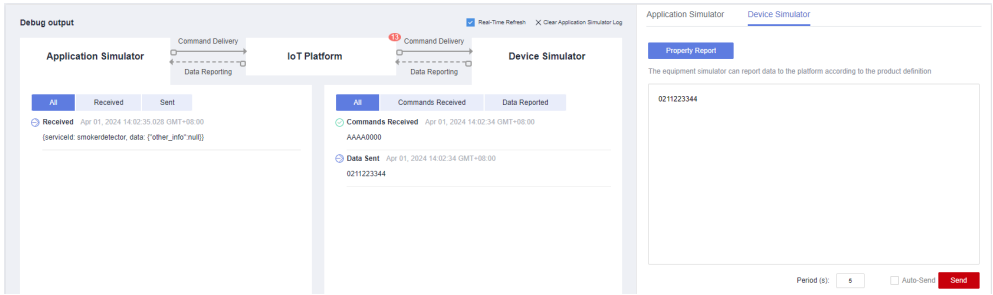


Step 4 Use the device simulator to report the description of the array type.

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.

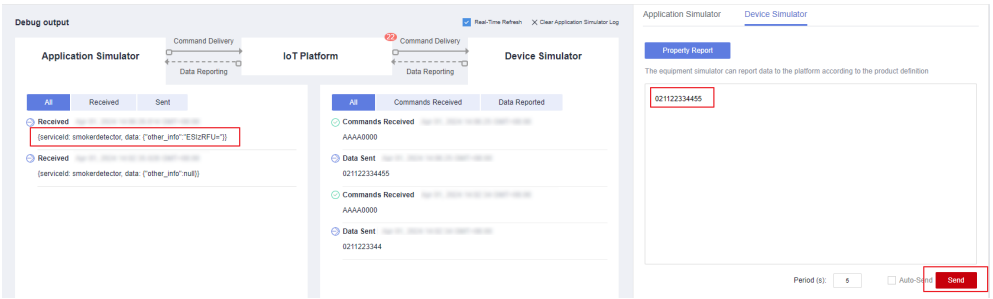
Figure 3-59 Simulating data reporting - other_info as array 1



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.

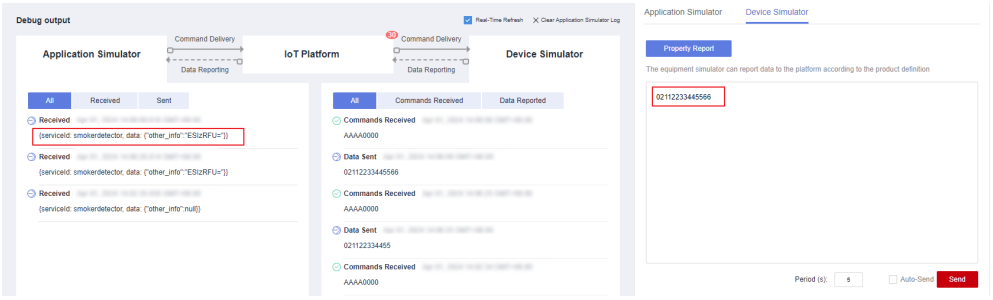
Figure 3-60 Simulating data reporting - other_info as array 2



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 3-61 Simulating data reporting - other_info as array 3

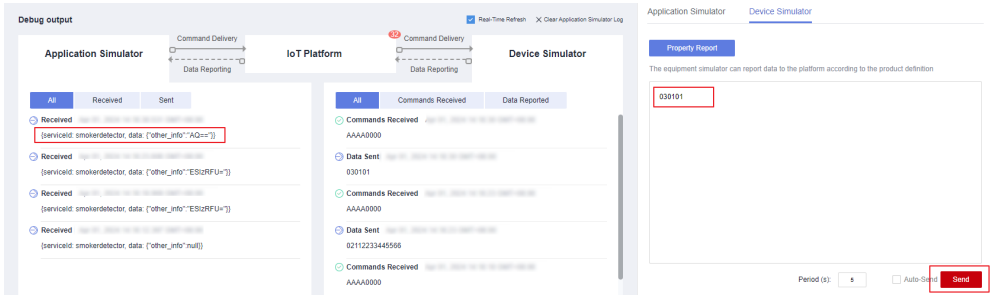


Step 5 Use the device simulator to report the description of the variable-length array type.

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.

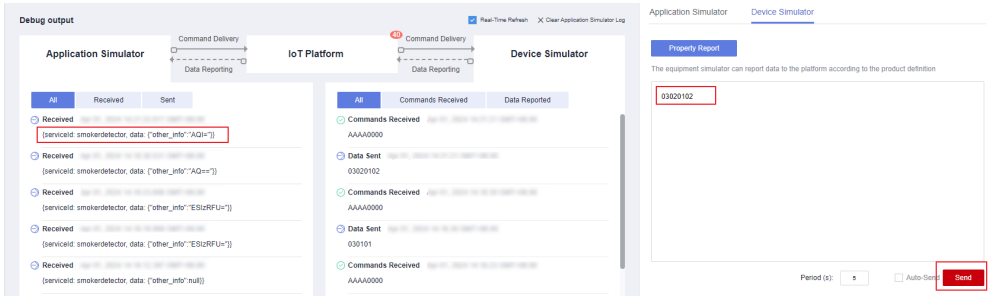
Figure 3-62 Simulating data reporting - other_info as variable-length array 1



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.

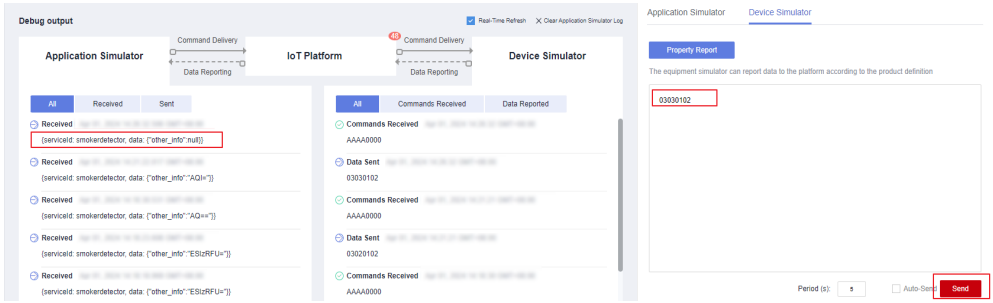
Figure 3-63 Simulating data reporting - other_info as variable-length array 2



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 3-64 Simulating data reporting - other_info as variable-length array 3

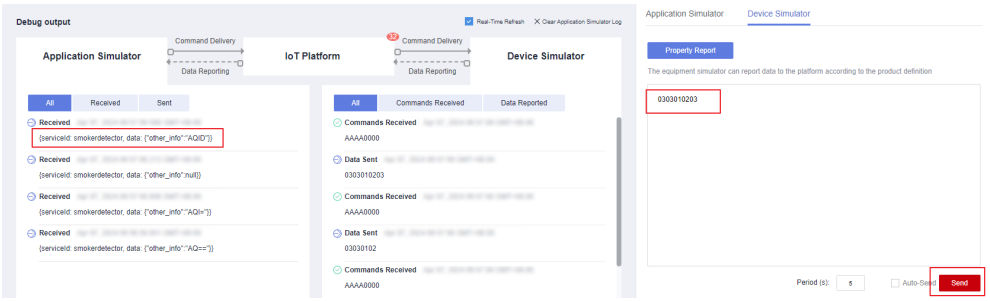


In the hexadecimal code stream example (0303010203), **03** indicates the **messageId** field and specifies that this message reports the description of the

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.

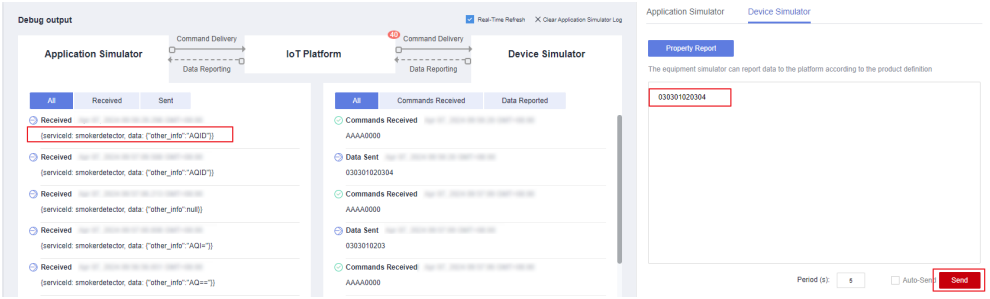
Figure 3-65 Simulating data reporting - other_info as variable-length array 4



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 3-66 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**.

3.4.3 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

- 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.

Example for a Smoke Detector

Scenario

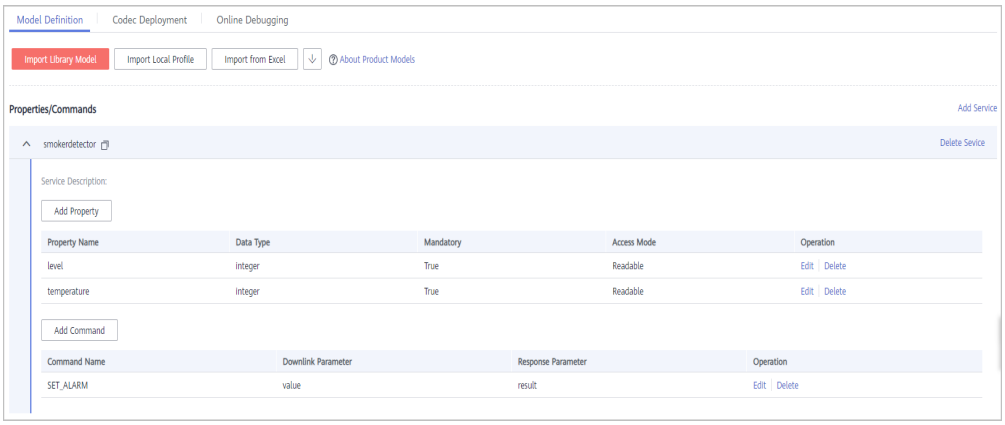
A smoke detector provides the following functions:

- 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 interface, but reporting simple binary data.

Product Model Definition

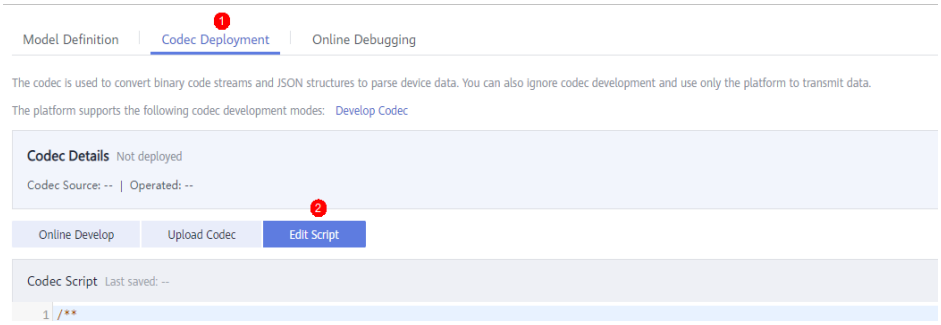
Define the product model on the product details page of the smoke detector.

- **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 the value **1** indicates that the alarm function is enabled.



Developing a Codec

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



Step 2 Compile 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 a 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 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 code stream into the JSON format used for 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 into 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
```



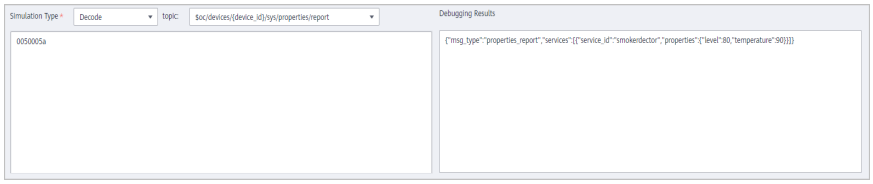
```
        }
        var paras_value = jsonObj.paras.value;
        payload = payload.concat(buffer_int16(paras_value)); // Set the command property value.
    }
   // Return the encoded binary data.
    return payload;
}
// Parse the message type based on the topic name.
function topicParse(topic) {
    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.

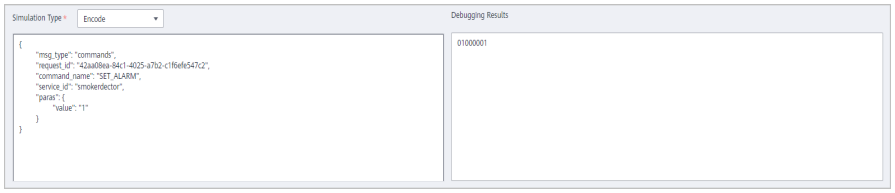
     

   - Check whether the debugging result meets the expectation. If the debugging result does not meet the expectation, modify the code and perform debugging again.

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**.

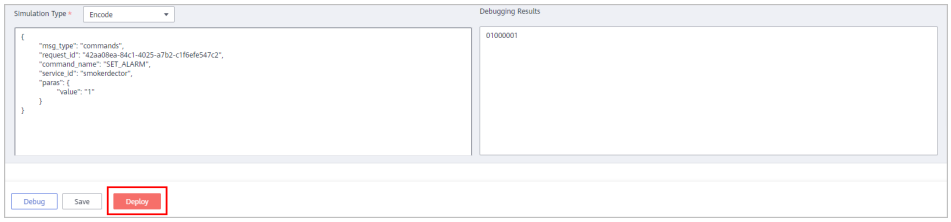
```
{
   "msg_type": "commands",
   "request_id": "42aa08ea-84c1-4025-a7b2-c1f6efe547c2",
   "command_name": "SET_ALARM",
   "service_id": "smokerdector",
   "paras": {
      "value": "1"
   }
}
```

   - 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.



   - 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.



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
The 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 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 code stream into the JSON format used for 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 code stream to the JSON format used by message reporting.
        //jsonObj = {"msg_type":"message_up","content":"hello"};
    }
    // Convert data to a character string in JSON format.
    return JSON.stringify(jsonObj);
}
/*
Sample data: When a command is delivered, JSON data on the IoT platform is encoded into binary code
streams 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 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 to a character 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 Definition

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"
   }]
}
```

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

ServiceProperty structure

Field	Mandatory	Type	Description
service_id	Yes	String	Identifies a service of the device.
properties	Yes	Object	Indicates service properties, which are defined in the product model associated with the device.
event_time	No	String	Indicates the UTC time when the device reports 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.

- Response for device property setting

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

Field	Mandatory	Type	Description

msg_type	Yes	String	Indicates the message type. The value is fixed at **properties_set_response**.
request_id	No	String	Uniquely identifies a 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	Indicates the command execution result. **0** indicates a successful execution, whereas other values indicate an execution failure. If this parameter is not carried, the execution is considered successful.
result_desc	No	String	Indicates the description of the response to the request for setting properties.

- Response for device 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"
      }
   ]
}
```

Field	Mandatory	Type	Description
msg_type	Yes	String	The value is fixed at **properties_get_response**.
request_id	No	String	Uniquely identifies a 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

Field	Mandatory	Type	Description
service_id	Yes	String	Identifies a service of the device.
properties	Yes	Object	Indicates service properties, which are defined in the product model associated with the device.
event_time	No	String	Indicates the UTC time when the device reports 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.

- Response for the platform to deliver a command

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

Field	Mandatory	Type	Description
msg_type	Yes	String	The value is fixed at **command_response**.
request_id	No	String	Uniquely identifies a 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	Indicates the command execution result. **0** indicates a successful execution, whereas other values indicate an execution failure. If this parameter is not carried, the execution is considered successful.
response_name	No	String	Indicates the response name, which is defined in the product model associated with the device.

paras	No	Object	Indicates the response parameters, which are defined in the product model associated with the device.

- Device message reporting

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

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

Data Encoding Format Definition

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. 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:

- Command delivery

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

Field	Mandatory	Type	Description
msg_type	Yes	String	The value is fixed at **commands**.
request_id	Yes	String	Uniquely identifies a request. The ID is delivered to the device through a topic.
service_id	No	String	Identifies a service of the device.
command_name	No	String	Indicates the device command name, which is defined in the product model associated with the device.
paras	No	Object	Indicates the command execution parameters, which are defined in the product model associated with the device.

- 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
          }
      }
   ]
}
```

Field	Mandatory	Type	Description
msg_type	Yes	String	The value is fixed at **properties_set**.
request_id	Yes	String	Uniquely identifies a 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>	Indicates a list of device service data.

ServiceProperty structure

Field	Mandatory	Type	Description
service_id	Yes	String	Identifies a service of the device.
properties	Yes	Object	Indicates service properties, which are defined in the product model associated with the device.

- Querying device properties

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

Field	Mandatory	Type	Description

msg_type	Yes	String	The value is fixed at **properties_get**.
request_id	Yes	String	Uniquely identifies a request. The ID is delivered to the device through a topic.
service_id	No	String	Identifies a service of the device.

- Response for property reporting (response to property reporting during NB-IoT device access)

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

Field	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.

- Message delivery

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

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

3.5 Online Debugging

Overview

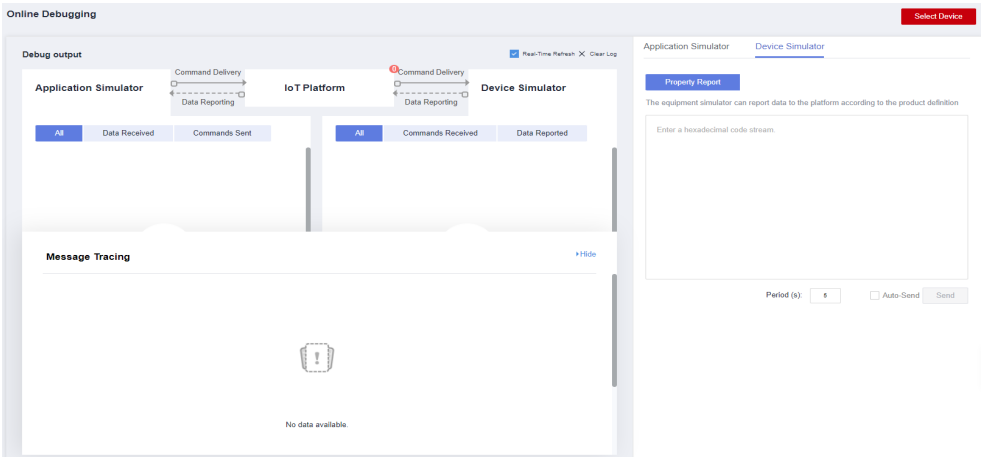
After the product model and codec are developed, the application can receive data reported by the device and deliver commands to the device through the IoT platform.

The 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.

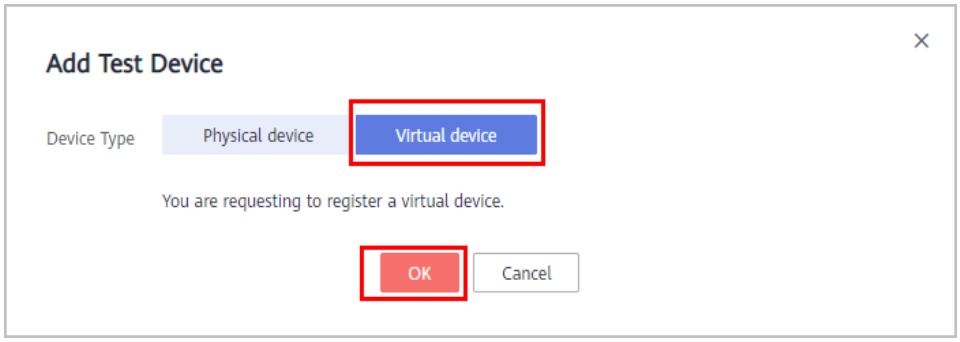
Debugging a Product by Using a Virtual Device

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 structure of the virtual device testing interface is as follows:

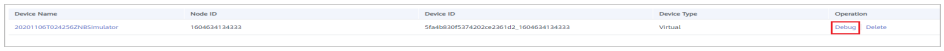


Step 1 On the product details page, click the **Online Debugging** tab and click **Add Test Device**.

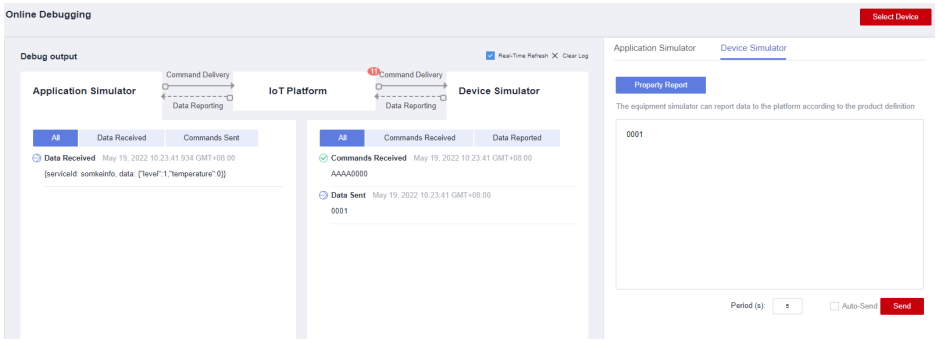
Step 2 In the **Add Test Device** dialog box, select **Virtual device** for **Device Type** and click **OK**. The virtual device name contains **DeviceSimulator**. Only one virtual device can be created for each product.



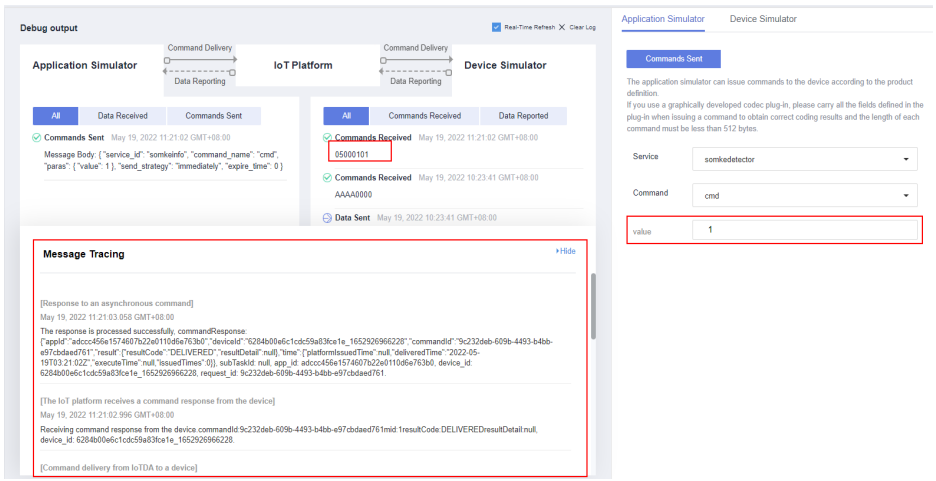
Step 3 In the device list, select the new virtual device and click **Debug** to enter the **Online Debugging** page.



Step 4 In **Device Simulator**, enter a hexadecimal code stream or JSON data (for example, enter a hexadecimal code stream) and click **Send**. View the data reporting result in **Application Simulator** and the processing logs of the IoT platform in **Message Tracing**.



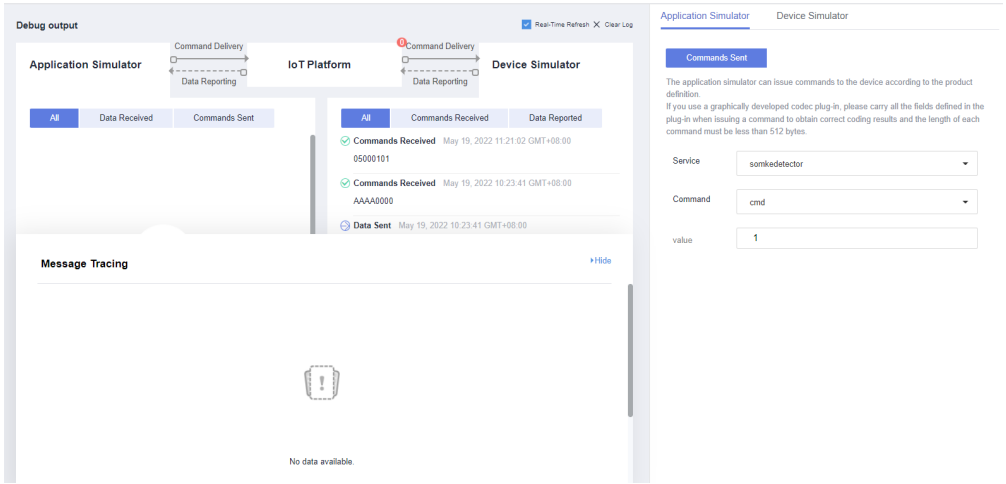
Step 5 Deliver a command in **Application Simulator**. View the received command (for example, a hexadecimal code stream) in **Device Simulator** and the processing logs of the IoT platform in **Message Tracing**.



----End

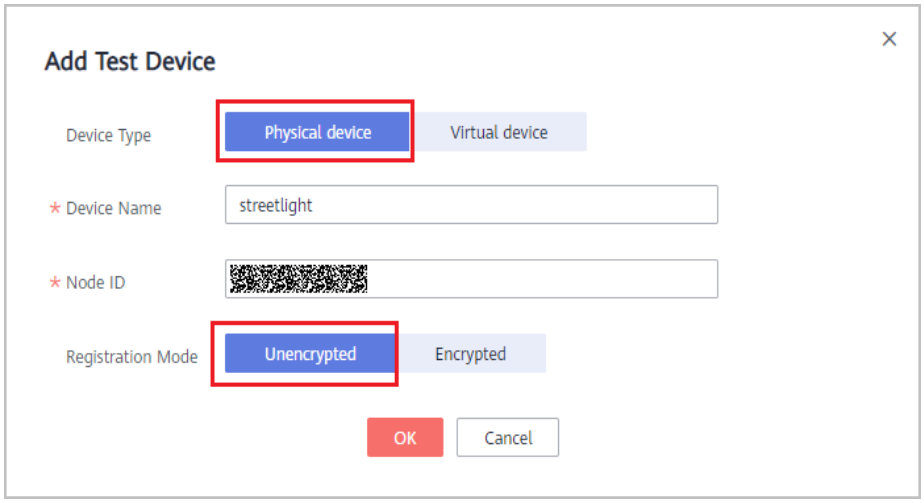
Debugging a Product by Using a Physical Device

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 structure of the physical device testing interface is as follows:



Step 1 On the product details page of the smoke detector, 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**.



Note: If DTLS is used for device access, set **Registration Mode** to **Encrypted** and keep the secret properly.

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.

Step 4 Simulate a scenario where a control command is remotely delivered. In **Application Simulator**, Set **Service** to **StreetLight**, **Command** to **SWITCH_LIGHT**, and **Command Value** to **ON**, and click **Send**. The street lamp is turned on.

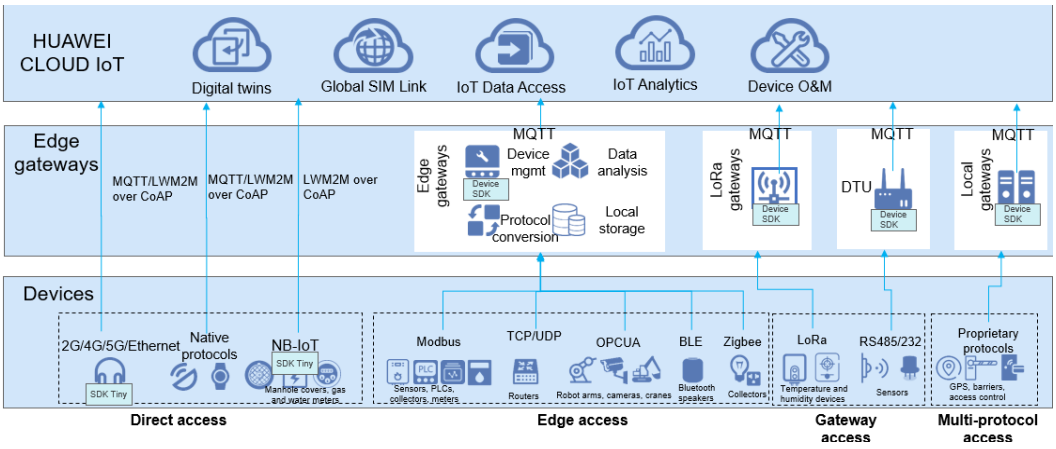
----End

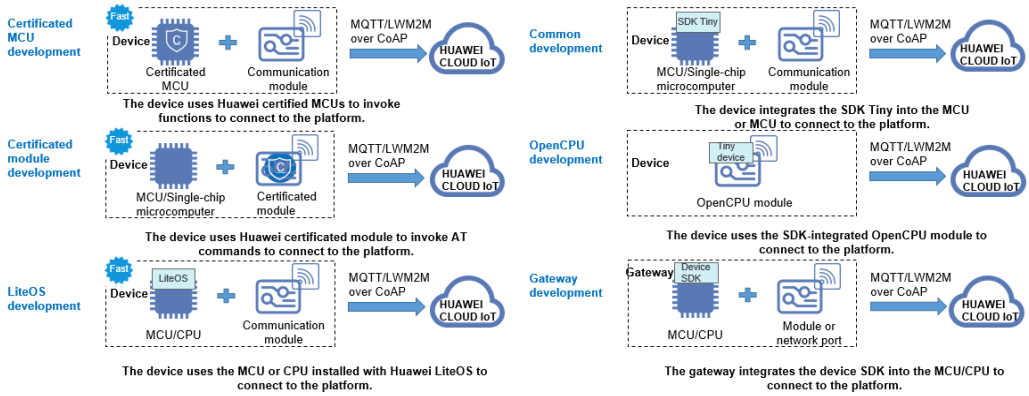
4 Development on the Device Side

4.1 Device Access Guide

Device Access Mode

The Huawei Cloud IoTDA provides multiple access modes to meet the requirements of device fleets in different access scenarios. You can select a proper development mode based on the device type.





Development Mode	Feature	Scenario	Difficulty Level
Certificated MCU development	The IoT Device SDK Tiny has been pre-integrated into the main control unit (MCU) and can call methods to connect to the platform.	Devices need to be quickly put into commercial use, with low R&D costs. Devices are connected to the platform directly, without using gateways.	★
Certificated module development	The IoT Device SDK Tiny has been pre-integrated into the module and can invoke AT commands to connect to the platform.	There are few MCU resources. Devices are connected to the platform directly, without using gateways.	★
LiteOS development	Devices run LiteOS that manages MCU resources. In addition, LiteOS has a built-in IoT Device SDK Tiny that can call functions to connect to the platform. This development mode shortens the device development duration and reduces the development difficulty.	No operating system is required. Devices are connected to the platform directly, without using gateways.	★★
Common development	The IoT Device SDK Tiny is integrated into the MCU and calls the SDK functions to connect to the platform. This type of call is more convenient than API access.	There is sufficient time for devices to put into commercial use, and the flash and RAM resources of the MCU meet the conditions for integrating the IoT Device SDK Tiny.	★★ ★
OpenCPU development	Use the MCU capability in the common module, and compile and run device applications on the OpenCPU.	Devices with a small size have high security requirements and need to be quickly put into commercial use.	★★ ★★
Gateway development	The IoT Device SDK is pre-integrated into the CPU or MPU and can call functions to connect to the platform.	Child devices connected to the platform using gateways.	★★ ★

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 SDK (Java)	Devices can connect to the platform by integrating the IoT Device SDK (Java). The demo provides the sample code for calling SDK APIs. For details, see **IoT Device SDK (Java)**.	**IoT Device SDK (Java)**
IoT Device SDK (C)	Devices can connect to the platform by integrating the IoT Device SDK (C). The demo provides the sample code for calling SDK APIs. For details, see **IoT Device SDK (C)**.	**IoT Device SDK (C)**
IoT Device SDK (C#)	Devices can connect to the platform by integrating the IoT Device SDK (C#). The demo provides the sample code for calling SDK APIs. For details, see **IoT Device SDK (C#)**.	**IoT Device SDK (C#)**
IoT Device SDK (Android)	Devices can connect to the platform by integrating the IoT Device SDK (Android). The demo provides the sample code for calling SDK APIs. For details, see **IoT Device SDK (Android)**.	**IoT Device SDK (Android)**
IoT Device SDK (Go)	Devices can connect to the platform by integrating the IoT Device SDK (Go). The demo provides the code sample for calling the SDK APIs. For details, see **IoT Device SDK (Go) User Guide**.	**IoT Device SDK (Go)**

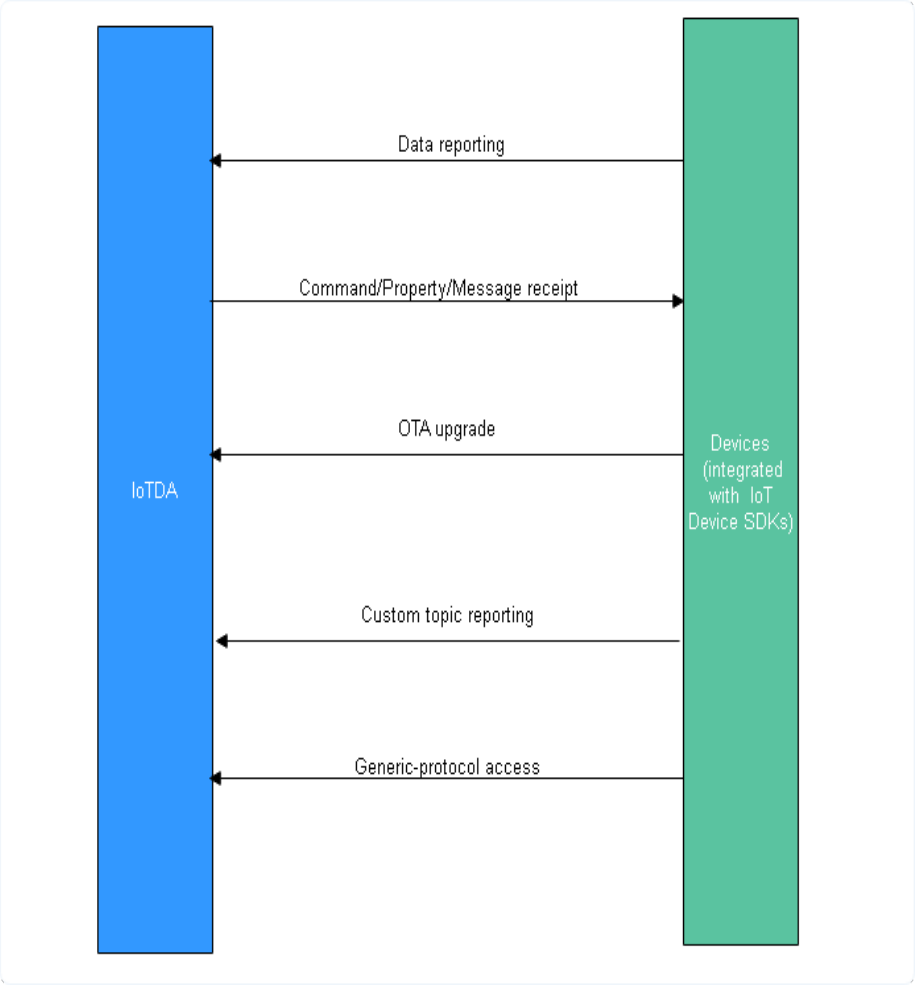
Resource Package	Description	Download Link
IoT Device SDK(Python)	Devices can connect to the platform by integrating the IoT Device SDK (Python). The demo provides the code sample for calling the SDK APIs. For details, see **IoT Device SDK (Python) Usage Guide**.	**IoT Device SDK(Python)**
IoT Device SDK Tiny (C)	Devices can connect to the platform by integrating the IoT Device SDK Tiny (C). The demo provides the sample code for calling SDK APIs. For details, see **IoT Device Tiny SDK (C)**.	**IoT Device SDK Tiny (C)**
Native MQTT or MQTTS access example	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)**
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**

Resource Package	Description	Download Link
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**

4.2 Using IoT Device SDKs for Access

4.2.1 Introduction to IoT Device SDKs

You can use Huawei IoT Device SDKs to quickly connect devices to the IoT platform. After being integrated with an IoT Device SDK, devices that support the TCP/IP protocol stack can directly communicate with the platform. Devices that do not support the TCP/IP protocol stack, such as Bluetooth and ZigBee devices, need to use a gateway integrated with the IoT Device SDK to communicate with the platform.



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 by calling the API **Creating a Device**.
3. Implement the functions demonstrated in the preceding figure, including reporting messages/properties, receiving commands/properties/messages, OTA upgrades, topic customization, and generic-protocol access (see **Developing a Protocol Conversion Gateway for Access of Generic-Protocol Devices**).

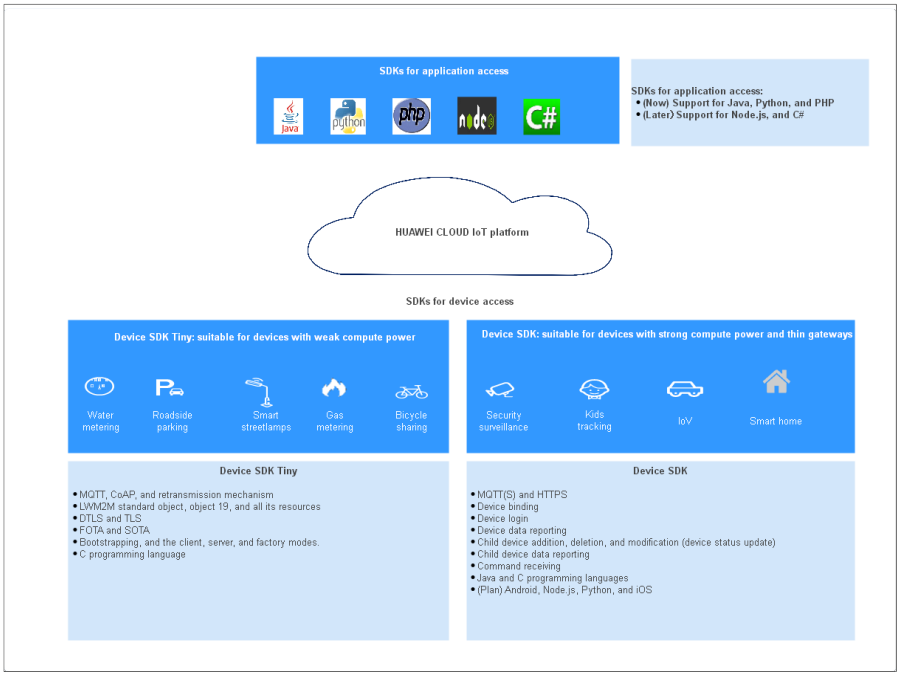
The platform provides two types of SDKs. The table below describes their differences.

SDK Type	Pre-integration Solution	IoT Protocols Supported
IoT Device SDK	Embedded devices with strong computing and storage capabilities, such as gateways and collectors	MQTT

SDK Type	Pre-integration Solution	IoT Protocols Supported
IoT Device SDK Tiny	Devices that have strict restrictions on power consumption, storage, and computing resources, such as single-chip microcomputer and modules	LwM2M over CoAP and MQTT

The table below describes hardware requirements for devices.

SDK	RAM Capacity	Flash Memory	CPU Frequency	OS Type	Programming Language
IoT Device SDK	> 4 MB	> 2 MB	> 200 MHz	C (Linux), Java (Linux/ Windows), C# (Windows), Android (Android), Go Community Edition (Linux/ Windows/Unix-like OS), and OpenHarmony	C, Java, C#, Android, and Go
IoT Device SDK Tiny	> 32 KB	> 128 KB	> 100 MHz	It adapts to LiteOS, Linux, macOS, and FreeRTOS. You can modify the SDK to **adapt to other environments**.	C



For details on the SDK usage, visit the following links:

- **IoT Device SDK (C)**
- **IoT Device SDK (Java)**
- **IoT Device SDK (C#)**
- **IoT Device SDK (Android)**
- **IoT Device SDK (Go Community Edition)**
- **IoT Device SDK Tiny (C)**
- **IoT Device SDK (Python)**

The following table shows the main function matrix of the SDK.

Table 4-1 SDK function matrix

Function	C	Java	C#	Android	Go	Python	C Tiny
Property reporting	√	√	√	√	√	√	√
Message reporting and delivery	√	√	√	√	√	√	√
Event reporting and delivery	√	√	√	√	√	√	√

Function	C	Java	C#	Android	Go	Python	C Tiny
Command delivery and response	√	√	√	√	√	√	√
Device shadow	√	√	√	√	√	√	√
OTA upgrade	√	√	√	√	√	√	√
bootstrap	√	√	√	√	√	√	√
Time synchronization	√	√	√	√	√	√	√
Gateway and child device management	√	√	√	√	√	√	√
Device-side Rules	√	×	√	×	×	×	√
Remote SSH	√	×	√	×	×	×	×
Anomaly detection	√	×	√	×	×	×	×
Device-cloud secure communication (soft bus)	√	×	√	×	×	×	×
M2M function	√	×	√	×	×	×	×

Function	C	Java	C#	Android	Go	Python	C Tiny
Generic-protocol access	√	√	√	√	×	√	×

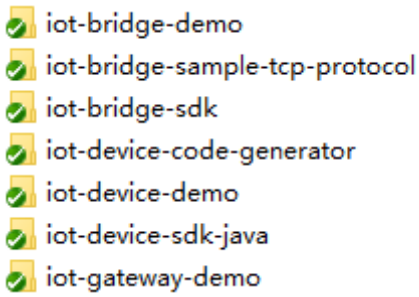
4.2.2 IoT Device SDK (Java)

Maven Reference

```
<dependencies>
   <dependency>
      <groupId>com.huaweicloud</groupId>
      <artifactId>iot-device-sdk-java</artifactId>
      <version>1.2.0</version>
   </dependency>
</dependencies>
```

Preparations

- Ensure that the JDK (version 1.8 or later) and Maven have been installed.
- **Download the SDK**. The project contains the following subprojects:

  iot-device-sdk-java: SDK code

  iot-device-demo: demo code for common directly connected devices

  iot-gateway-demo: demo code for gateways

  iot-bridge-sdk: SDK code for the bridge

  iot-bridge-demo: demo code for the bridge, which is used to bridge a TCP device to the platform

  iot-bridge-sample-tcp-protocol: sample code of a child device using TCP to connect to a bridge

  iot-device-code-generator: device code generator, which can automatically generate device code for different product models
- Go to the SDK root directory and run the **mvn install** command to build and install the SDK.

Creating a Product

A smokeDetector product model is provided to help you understand the product model. This smoke detector can report the smoke density, temperature, humidity,

and smoke alarms, and execute the ring alarm command. The following uses the smoke detector as an example to introduce the procedures of message reporting and property reporting.

Step 1 Access the **IoTDA** service page and click **Access Console**. Click the target instance card. Check and save the MQTTS device access domain name.

Step 2 Choose **Products** in the navigation pane and click **Create Product**.

Step 3 Set the parameters as prompted and click **OK**.

Set Basic Info	
Resource Space	The platform automatically allocates the created product to the default resource space. If you want to allocate the product to another resource space, select the resource space from the drop-down list box. If a **resource space** does not exist, create it first.
Product Name	Customize the product name. The name can contain letters, numbers, underscores (_), and hyphens (-).
Protocol	Select **MQTT**.
Data Type	Select **JSON**.
Device Type Selection	Select **Custom**.
Device Type	Select **smokeDetector**.
Advanced Settings	
Product ID	Leave this parameter blank.
Description	Set this parameter based on service requirements.

----End

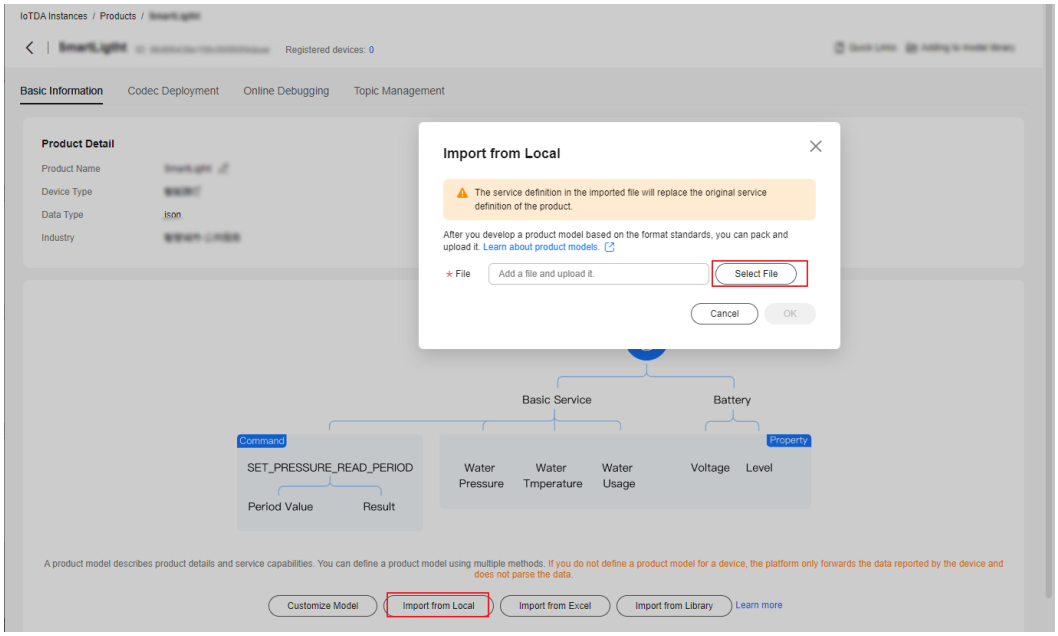
Uploading a Product Model

Step 1 Download the **smokeDetector product model file**.

Step 2 Click the name of the product created in **3** to access its details.

Step 3 On the **Basic Information** tab page, click **Import from Local** to upload the product model file obtained in **1**.

Figure 4-1 Product - Uploading a product model



----End

Registering a Device

Step 1 In the navigation pane, choose **Devices** > **All Devices**, and click **Register Device**.

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

Parameter	Description
Resource Space	Ensure that the device and the product created in **3** belong to the same resource space.
Product	Select the product created in **3**.
Node ID	This parameter specifies the unique physical identifier of the device. The value can be customized and consists of letters and numbers.
Device Name	Customize the device name.
Authentication Type	Select **Secret**.
Secret	Customize the device secret. If this parameter is not set, the platform automatically generates a secret.

After the device is registered, save the node ID, device ID, and secret.

----End

Initializing a Device

1. Enter the device ID and secret obtained in **Registering a Device** and the device connection information obtained in **1**. The format is ***ssl://Domain name:Port*** or ***ssl://IP address:Port***.

```
// Obtaining the certificate path: Load the CA certificate of the IoT platform and use the default ca.jks of the SDK for server verification.
    URL resource = MessageSample.class.getClassLoader().getResource("ca.jks");
    File file = new File(resource.getPath());
    //For example, modify the following parameters in MessageSample.java in the iot-device-demo file:
    IoTDevice device = new IoTDevice("ssl://Domain name:8883",
            "5e06bfee334dd4f33759f5b3_demo", "mysecret", file);
```

CAUTION

All files that involve device IDs and passwords must be modified accordingly.

2. Establish a connection. Call **init** of the IoT Device SDK. The thread is blocked until a result is returned. If the connection is established, **0** is returned.

```
    if (device.init() != 0) {
        return;
    }
```

If the connection is successful, information similar to the following is displayed:

```
2023-07-17 17:22:59 INFO MqttConnection:105 - Mqtt client connected. address :ssl://Domain name:8883
```

3. After the device is created and connected, it can be used for communication. You can call **getClient** of the IoT Device SDK to obtain the device client. The client provides communication APIs for processing messages, properties, and commands.

Reporting a Message

Message reporting is the process in which a device reports messages to the platform.

1. Call **getClient** of the IoT Device SDK to obtain the client from the device.
2. Call **reportDeviceMessage** to enable the client to report a device message. In the sample below, messages are reported periodically.

```
    while (true) {
        device.getClient().reportDeviceMessage(new DeviceMessage("hello"), new ActionListener() {
            @Override
            public void onSuccess(Object context) {
                log.info("reportDeviceMessage ok");
            }

            @Override
            public void onFailure(Object context, Throwable var2) {
                log.error("reportDeviceMessage fail: " + var2);
            }
        });

        // Report a message using a custom topic, which must be configured on the platform first.
        String topic = "$oc/devices/" + device.getDeviceId() + "/user/wpy";
        device.getClient().publishRawMessage(new RawMessage(topic, "hello raw message "),
            new ActionListener() {
                @Override
                public void onSuccess(Object context) {
                    log.info("publishRawMessage ok: ");
```



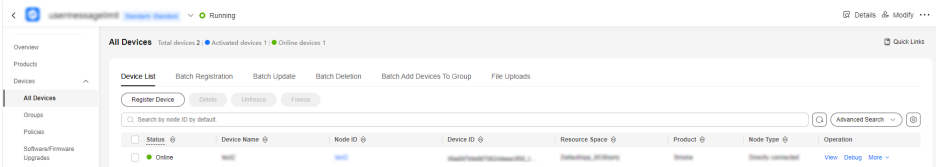
```
            }

            @Override
            public void onFailure(Object context, Throwable var2) {
                log.error("publishRawMessage fail: " + var2);
            }
        });

        Thread.sleep(5000);
    }
```

3. Replace the device parameters with the actual values in the **main** function of the **MessageSample** class, and run this class. Then view the logs about successful connection and message reporting.

```
2024-04-16 16:43:09  INFO AbstractService:103 - create device, the deviceId is
5e06bfee334dd4f33759f5b3_demo
2024-04-16 16:43:09 INFO MqttConnection:233 - try to connect to ssl://Domain name: 8883
2024-04-16 16:43:10 INFO MqttConnection:257 - connect success, the uri is ssl://Domain name: 8883
2024-04-16 16:43:11  INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/sys/events/up, msg =
{"object_device_id":"5e06bfee334dd4f33759f5b3_demo","services":[{"paras":
{"type":"DEVICE_STATUS","content":"connect
success","timestamp":"1713256990817"},"service_id":"$log","event_type":"log_report","event_time":"20
240416T084310Z","event_id":null}]}
2024-04-16 16:43:11 INFO MqttConnection:140 - Mqtt client connected. address is ssl://Domain
name: 8883
2024-04-16 16:43:11  INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/sys/events/up, msg =
{"object_device_id":"5e06bfee334dd4f33759f5b3_demo","services":[{"paras":
{"device_sdk_version":"JAVA_v1.2.0","fw_version":null,"sw_version":null},"service_id":"$sdk_info","event
_type":"sdk_info_report","event_time":"20240416T084311Z","event_id":null}]}
2024-04-16 16:43:11 INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo /sys/events/up, msg = {"object_device_id":
"5e06bfee334dd4f33759f5b3_demo ","services": [{"paras":
{"type":"DEVICE_STATUS","content":"connect complete, the url is ssl://Domain name:
8883","timestamp":"1713256991263"},"service_id":"$log","event_type":"log_report","event_time":"2024
0416T084311Z","event_id":null}]}
2024-04-16 16:43:11  INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/sys/messages/up, msg =
{"name":null,"id":null,"content":"hello","object_device_id":null}
2024-04-16 16:43:11  INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/user/wpy, msg =  hello raw message
2024-04-16 16:43:11  INFO MessageSample:98 - reportDeviceMessage ok
2024-04-16 16:43:11  INFO MessageSample:113 - publishRawMessage ok:
```

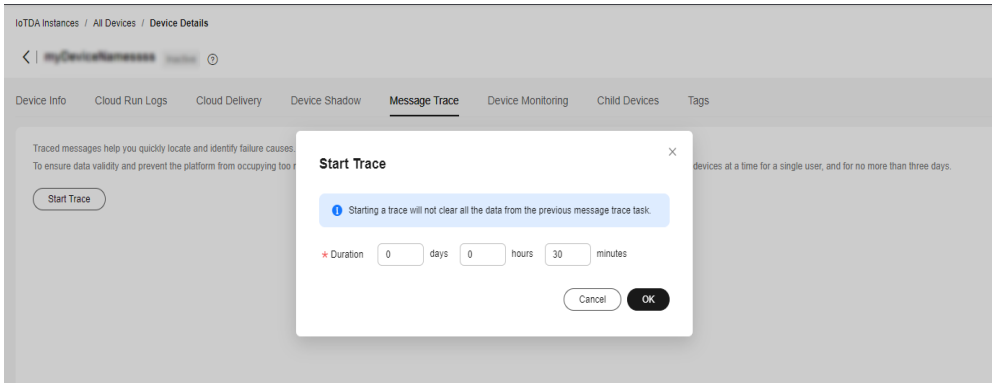
4. On the IoTDA console, choose **Devices** > **All Devices** and check whether the device is online.

Figure 4-2 Device list - Device online status



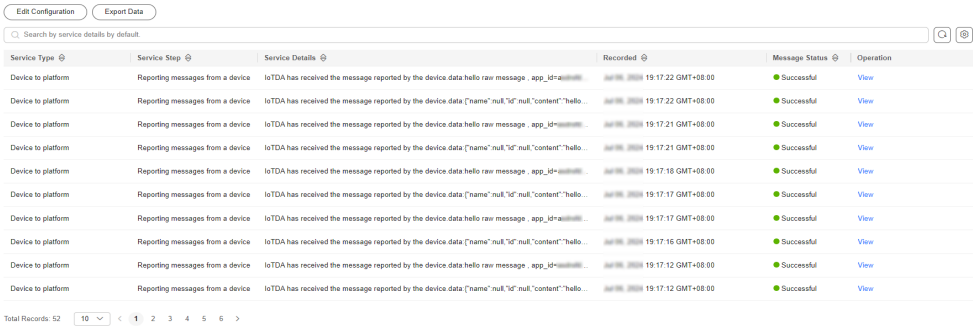
5. Select the device, click **View**, and enable message trace on the device details page.

Figure 4-3 Message tracing - Starting message tracing



6. View the messages received by the platform.

Figure 4-4 Message tracing - Viewing device_sdk_java tracing result



Note: Message trace may be delayed. If no data is displayed, wait for a while and refresh the page.

Reporting Properties

Open the **PropertySample** class. In this example, the **alarm**, **temperature**, **humidity**, and **smokeConcentration** properties are periodically reported to the platform.

```
    // Report properties periodically.
      while (true) {

          Map<String ,Object> json = new HashMap<>();
          Random rand = new Random();

          // Set properties based on the product model.
          json.put("alarm", 1);
          json.put("temperature", rand.nextFloat()*100.0f);
          json.put("humidity", rand.nextFloat()*100.0f);
          json.put("smokeConcentration", rand.nextFloat() * 100.0f);

          ServiceProperty serviceProperty = new ServiceProperty();
          serviceProperty.setProperties(json);
          serviceProperty.setServiceId("smokeDetector");// The serviceId must the consistent with that
defined in the product model.


            device.getClient().reportProperties(Arrays.asList(serviceProperty), new ActionListener() {
             @Override
             public void onSuccess(Object context) {
                log.info("pubMessage success" );
             }
```



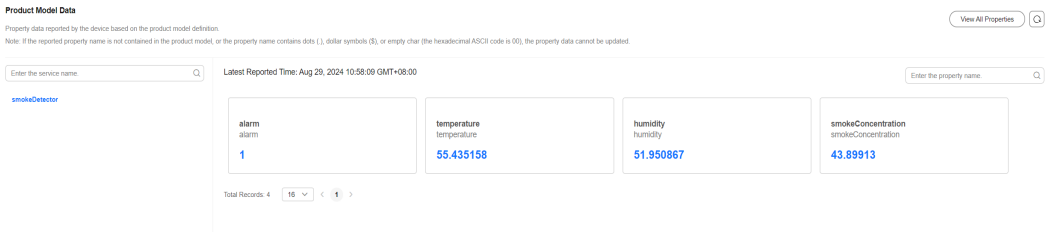
```
            @Override
            public void onFailure(Object context, Throwable var2) {
                log.error("reportProperties failed" + var2.toString());
            }
        });

        Thread.sleep(10000);
    }
}
```

Modify the **main** function of the **PropertySample** class and run this class. Then view the logs about successful property reporting.

```
2024-04-17 15:38:37  INFO AbstractService:103 - create device, the deviceId is
5e06bfee334dd4f33759f5b3_demo
2024-04-17 15:38:37 INFO MqttConnection:233 - try to connect to ssl://Domain name: 8883
2024-04-17 15:38:38 INFO MqttConnection:257 - connect success, the uri is ssl://Domain name: 8883
2024-04-17 15:38:38  INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/sys/events/up, msg =
{"object_device_id":"661e35467bdccc0126d1a595_feng-sdk-test3","services":[{"paras":
{"type":"DEVICE_STATUS","content":"connect
success","timestamp":"1713339518043"},"service_id":"$log","event_type":"log_report","event_time":"2024041
7T073838Z","event_id":null}]}
2024-04-17 15:38:38 INFO MqttConnection:140 - Mqtt client connected. address is ssl://Domain name: 8883
2024-04-17 15:38:38  INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/sys/events/up, msg =
{"object_device_id":"661e35467bdccc0126d1a595_feng-sdk-test3","services":[{"paras":
{"device_sdk_version":"JAVA_v1.2.0","fw_version":null,"sw_version":null},"service_id":"$sdk_info","event_type"
:"sdk_info_report","event_time":"20240417T073838Z","event_id":null}]}
2024-04-17 15:38:38 INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo /sys/events/up, msg = {"object_device_id":
"5e06bfee334dd4f33759f5b3_demo ","services": [{"paras":{"type":"DEVICE_STATUS","content":"connect
complete, the url is ssl://Domain
name :8883","timestamp":"1713339518464"},"service_id":"$log","event_type":"log_report","event_time":"202
40417T073838Z","event_id":null}]}
2024-04-17 15:38:38  INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/sys/properties/report, msg =  {"services":[{"properties":
{"alarm":1,"temperature":55.435158,"humidity":51.950867,"smokeConcentration":43.89913},"service_id":"sm
okeDetector","event_time":null}]}
2024-04-17 15:38:38  INFO PropertySample:144 - pubMessage success
```

The latest property values are displayed on the device details page of the platform.

Figure 4-5 Product model - Property reporting



Reading and Writing Properties

Call the **setPropertyListener** method of the client to set the property callback. In **PropertySample**, the property reading/writing API is implemented.

Property reading: Only the **alarm** property can be written.

Property reading: Assemble the local property value based on the API format.

```
        device.getClient().setPropertyListener(new PropertyListener() {
```



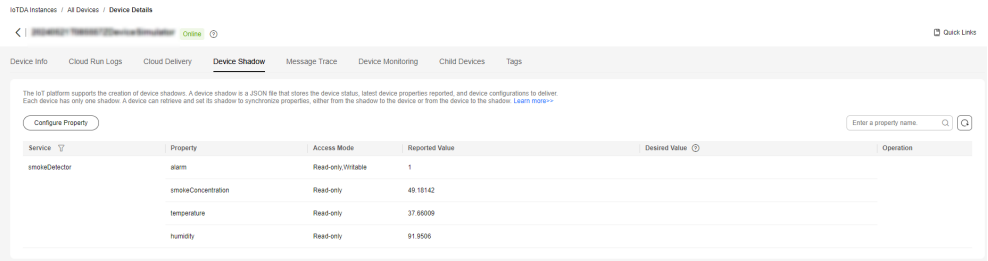
```
        // Process property writing.
        @Override
        public void onPropertiesSet(String requestId, List<ServiceProperty> services) {
            // Traverse services.
            for (ServiceProperty serviceProperty : services) {

                log.info("OnPropertiesSet, serviceId is {}", serviceProperty.getServiceId());

                // Traverse properties.
                for (String name : serviceProperty.getProperties().keySet()) {
                    log.info("property name is {}", name);
                    log.info("set property value is {}", serviceProperty.getProperties().get(name));
                }

            }
            // Change the local property value.
            device.getClient().respondPropsSet(requestId, IotResult.SUCCESS);
        }

        /**
         * Process property reading. In most scenarios, you can directly read the device shadow on the
platform, so this interface does not need to be implemented.
         * To read device properties in real time, implement this method.
         */
        @Override
        public void onPropertiesGet(String requestId, String serviceId) {
            log.info("OnPropertiesGet, the serviceId is {}", serviceId);
            Map<String, Object> json = new HashMap<>();
            Random rand = new SecureRandom();
            json.put("alarm", 1);
            json.put("temperature", rand.nextFloat() * 100.0f);
            json.put("humidity", rand.nextFloat() * 100.0f);
            json.put("smokeConcentration", rand.nextFloat() * 100.0f);

            ServiceProperty serviceProperty = new ServiceProperty();
            serviceProperty.setProperties(json);
            serviceProperty.setServiceId("smokeDetector");

            device.getClient().respondPropsGet(requestId, Arrays.asList(serviceProperty));
        }
    });
```

Note:

1. The property reading/writing API must call **respondPropsGet** and **respondPropsSet** to report the operation result.
2. If the device does not allow the platform to proactively read data from the device, **onPropertiesGet** can be left not implemented.

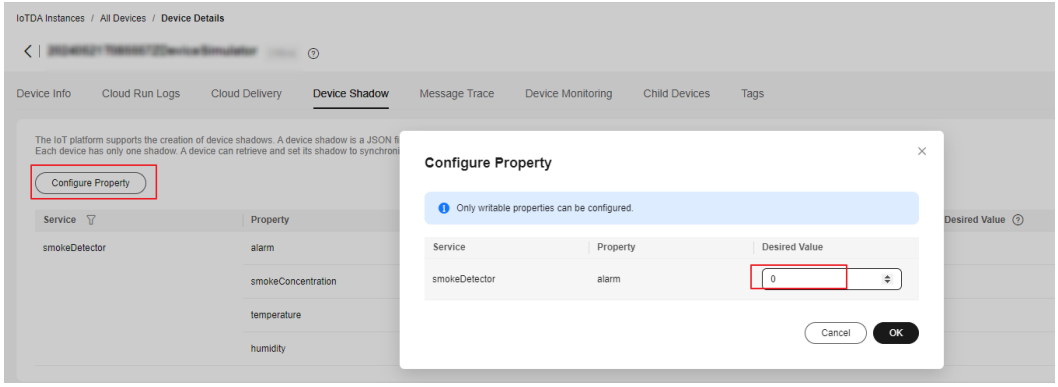
Run the **PropertySample** class and check whether the value of the **alarm** property is **1** on the **Device Shadow** tab page.

Figure 4-6 Device shadow - Viewing property (Alarm)



Change the value of the **alarm** property to **0**.

Figure 4-7 Device shadow - Configuring property (alarm)



In the device logs, the value of **alarm** is **0**.

```
INFO MqttConnection:66 - messageArrived topic =  $oc/devices/                    _demo/sys/properties/set/reque
INFO PropertySample:53 - OnPropertiesSet, serviceId =  smokeDetector
INFO PropertySample:57 - property name = alarm
INFO PropertySample:58 - set property value = 0
```

Delivering a Command

You can set a command listener to receive commands delivered by the platform. The callback needs to process the commands and report responses.

The **CommandSample** class prints commands after receiving them and calls **respondCommand** to report the responses.

```
device.getClient().setCommandListener(new CommandListener() {
        @Override
        public void onCommand(String requestId, String serviceId, String commandName, Map<String,
Object> paras) {
            log.info("onCommand, serviceId = {}", serviceId);
            log.info("onCommand , name = {}", commandName);
            log.info("onCommand, paras =  {}", paras.toString());

            // Process the command.

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

    });
```

Run the **CommandSample** class and deliver a command on the platform. In the command, set **serviceId** to **smokeDetector**, **name** to **ringAlarm**, and **paras** to **duration=20**.

The log shows that the device receives the command and reports a response.

```
INFO MqttConnection:66 - messageArrived topic =  $oc/devices/test_testDevice/sys/commands/request_id=4, msg = {"paras":{"duration":20},"service_id":"smo
INFO CommandSample:62 - onCommand, serviceId = smokeDetector
INFO CommandSample:63 - onCommand , name = ringAlarm
INFO CommandSample:64 - onCommand, paras =  {duration=20}
INFO MqttConnection:213 - publish message topic =  $oc/devices/test_testDevice/sys/commands/response/request_id=4, msg = {"paras":null,"result_code":0,"
```


Object-oriented Programming

Calling device client APIs to communicate with the platform is flexible but requires you to properly configure each API.

The SDK provides a simpler method, object-oriented programming. You can use the product model capabilities provided by the SDK to define device services and call the property reading/writing API to access the device services. In this way, the SDK can automatically communicate with the platform to synchronize properties and call commands.

Object-oriented programming simplifies the complexity of device code and enables you to focus only on services rather than the communications with the platform. This method is much easier than calling client APIs and suitable for most scenarios.

The following uses **smokeDetector** to demonstrate the process of object-oriented programming.

1. Define the service class and properties based on the product model. (If there are multiple services, define multiple service classes.)

```
public static class SmokeDetectorService extends AbstractService {

    // Define properties based on the product model. Ensure that the device name and type are the same as those in the product model. writeable indicates whether the property can be written, and name indicates the property name.
    @Property(name = "alarm", writeable = true)
    int smokeAlarm = 1;

    @Property(name = "smokeConcentration", writeable = false)
    float concentration = 0.0f;

    @Property(writeable = false)
    int humidity;

    @Property(writeable = false)
    float temperature;
```

   @Property indicates a property. You can use **name** to specify a property name. If no property name is specified, the field name is used.

   You can add **writeable** to a property to control permissions on it. If the property is read-only, add **writeable = false**. If **writeable** is not added, the property can be read and written.

2. Define service commands. The SDK automatically calls the service commands when the device receives commands from the platform.

   The type of input parameters and return values for APIs cannot be changed. Otherwise, a runtime error occurs.

   The following code defines a ring alarm command named **ringAlarm**. The delivered parameter is **duration**, which indicates the duration of the ringing alarm.

```
// Define the command. The type of input parameters and return values for APIs cannot be changed. Otherwise, a runtime error occurs.
    @DeviceCommand(name = "ringAlarm")
    public CommandRsp alarm(Map<String, Object> paras) {
        int duration = (int) paras.get("duration");
        log.info("ringAlarm  duration = " + duration);
        return new CommandRsp(0);
    }
```

3. Define the getter and setter methods.

- The device automatically calls the **getter** method after receiving the commands for querying and reporting properties from the platform. The getter method reads device properties from the sensor in real time or from the local cache.
- The device automatically calls the setter method after receiving the commands for setting properties from the platform. The setter method updates the local values of the device. If a property is not writable, leave the setter method not implemented.

```
// Ensure that the names of the setter and getter methods comply with the JavaBean specifications so that the APIs can be automatically called by the SDK.
    public int getHumidity() {

        // Simulate the action of reading data from the sensor.
        humidity = new Random().nextInt(100);
        return humidity;
    }

    public void setHumidity(int humidity) {
        // You do not need to implement this method for read-only fields.
    }

    public float getTemperature() {

        // Simulate the action of reading data from the sensor.
        temperature = new Random().nextInt(100);
        return temperature;
    }

    public void setTemperature(float temperature) {
        // You do not need to implement this method for read-only fields.
    }

    public float getConcentration() {

        // Simulate the action of reading data from the sensor.
        concentration = new Random().nextFloat()*100.0f;
        return concentration;
    }

    public void setConcentration(float concentration) {
        // You do not need to implement this method for read-only fields.
    }

    public int getSmokeAlarm() {
        return smokeAlarm;
    }

    public void setSmokeAlarm(int smokeAlarm) {

        this.smokeAlarm = smokeAlarm;
        if (smokeAlarm == 0){
            log.info("alarm is cleared by app");
        }
    }
```

4. Create a service instance in the **main** function and add the service instance to the device.

```
    // Create a device.
    IoTDevice device = new IoTDevice(serverUri, deviceId, secret);

    // Create a device service.
    SmokeDetectorService smokeDetectorService = new SmokeDetectorService();
    device.addService("smokeDetector", smokeDetectorService);

    if (device.init() != 0) {
```



```
        return;
    }
```

5. Enable periodic property reporting.

```
// Enable periodic property reporting.
smokeDetectorService.enableAutoReport(10000);
```

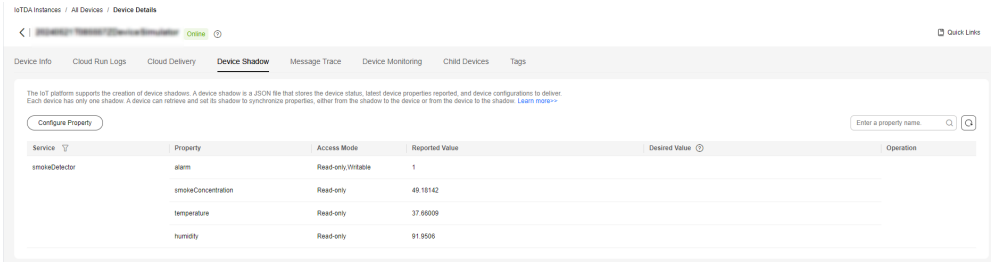
If you do not want to report properties periodically, you can call **firePropertiesChanged** to manually report them.

Run the **SmokeDetector** class to view the logs about property reporting.

```
INFO MqttConnection:140 - try to connect to ssl://[redacted].myhuaweicloud.com:8883
INFO MqttConnection:147 - connect success ssl://[redacted].myhuaweicloud.com:8883
INFO MqttConnection:87 - Mqtt client connected. address :ssl://[redacted].myhuaweicloud.com:8883
connect ok
INFO MqttConnection:213 - publish message topic =  $oc/devices/5e06bfee334dd4f33759f5b3_demo/sys/properties/report, msg = {"service
```

View the device shadow on the platform.

Figure 4-8 Device shadow - Viewing property (Alarm)



Modify the **alarm** property on the platform and view the device logs about property modification.

```
INFO MqttConnection:66 - messageArrived topic =  $oc/devices/test_testDevice/sys/properties/set/request_id=2, msg = {"services":[{"pro
INFO AbstractService:187 - write property ok:alarm
```

Deliver the **ringAlarm** command on the platform.

View the logs about calling the **ringAlarm** command and reporting a response.

```
INFO MqttConnection:66 - messageArrived topic =  $oc/devices/test_testDevice/sys/commands/request_id=1, msg = {"paras":{"duration":20},
INFO DeviceServiceSample$SmokeDetectorService:53 - ringAlarm  duration = 20
INFO MqttConnection:213 - publish message topic =  $oc/devices/test_testDevice/sys/commands/response/request_id=1, msg = {"paras":null,
```

Using the Code Generator

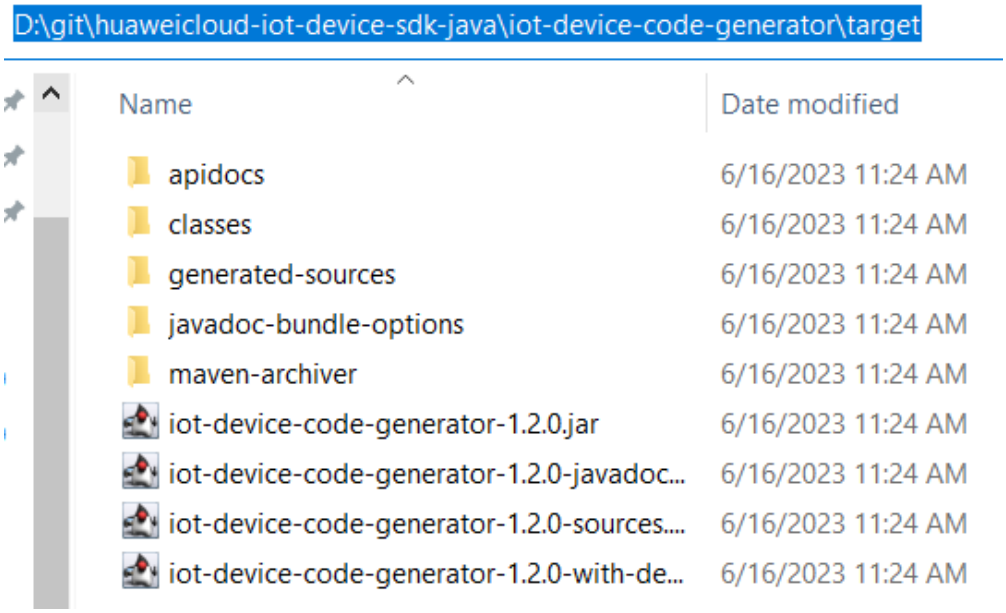
The SDK provides a code generator, which allows you to automatically generate a device code framework only using a product model. The code generator parses the product model, generates a service class for each service defined in the model, and generates a device main class based on the service classes. In addition, the code generator creates a device and registers a service instance in the **main** function.

To use the code generator to generate device code, proceed as follows:

1. Download the **huaweicloud-iot-device-sdk-java** project, decompress it, go to the **huaweicloud-iot-device-sdk-java** directory, and run the **mvn install** command.


```
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary for huaweicloud iot device sdk project for java 1.2.0:
[INFO]
[INFO] huaweicloud iot device sdk project for java ........ SUCCESS [  0.802 s]
[INFO] iot-device-sdk-java ................................ SUCCESS [  3.976 s]
[INFO] iot-device-demo .................................... SUCCESS [  4.112 s]
[INFO] iot-bridge-sdk ..................................... SUCCESS [ 17.187 s]
[INFO] iot-bridge-demo .................................... SUCCESS [  4.168 s]
[INFO] iot-gateway-demo ................................... SUCCESS [  2.852 s]
[INFO] iot-device-code-generator .......................... SUCCESS [  2.658 s]
[INFO] iot-bridge-sample-tcp-protocol ..................... SUCCESS [  4.122 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  39.978 s
[INFO] Finished at: 2023-06-16T11:25:00+08:00
[INFO] ------------------------------------------------------------------------
```

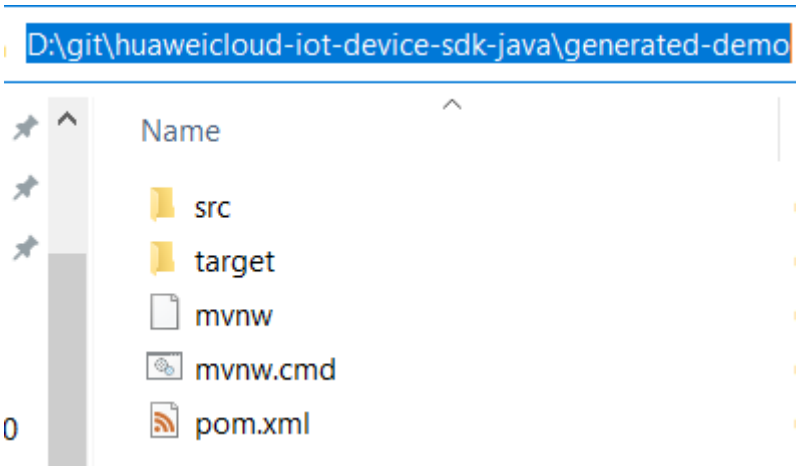
2. Check whether an executable JAR package is generated in the **target** folder of **iot-device-code-generator**.



3. Save the product model to a local directory. For example, save the **smokeDetector.zip** file to disk D.
4. Access the SDK root directory and run the **java -jar .\iot-device-code-generator\target\iot-device-code-generator-1.2.0-with-deps.jar D:\smokeDetector.zip** command.

```
PS D:\git\huaweicloud-iot-device-sdk-java> java -jar .\iot-device-code-generator\target\iot-device-code-generator-1.2.0-with-deps.jar D:\smokeDetector.zip
2023-06-16 11:30:47  INFO DeviceCodeGenerator:147 - the file generation path is :D:\git\huaweicloud-iot-device-sdk-java\generated-demo\src\main\java\com\huaweicloud\sdk\iot\device\demo\smokeDetectorService.java
2023-06-16 11:30:47  INFO DeviceCodeGenerator:73 - demo code generated to: D:\git\huaweicloud-iot-device-sdk-java\generated-demo
```

5. Check whether the **generated-demo** package is generated in the **huaweicloud-iot-device-sdk-java** directory.



The device code is generated.

To compile the generated code, proceed as follows:

1. Go to the **huaweicloud-iot-device-sdk-java\generated-demo** directory, and run the **mvn install** command to generate a JAR package in the **target** folder.

```
PS D:\git\huaweicloud-iot-device-sdk-java> cd .\generated-demo\
PS D:\git\huaweicloud-iot-device-sdk-java\generated-demo> mvn install
```

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  4.386 s
[INFO] Finished at: 2023-06-16T11:31:47+08:00
[INFO] ------------------------------------------------------------------------
PS D:\git\huaweicloud-iot-device-sdk-java\generated-demo> dir target


    Directory: D:\git\huaweicloud-iot-device-sdk-java\generated-demo\target


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         6/16/2023  11:31 AM                apidocs
d-----         6/16/2023  11:31 AM                classes
d-----         6/16/2023  11:31 AM                generated-sources
d-----         6/16/2023  11:31 AM                javadoc-bundle-options
d-----         6/16/2023  11:31 AM                maven-archiver
-a----         6/16/2023  11:31 AM          29924 iot-device-demo-ganerated-1.2.0-javado
                                                  c.jar
-a----         6/16/2023  11:31 AM           6728 iot-device-demo-ganerated-1.2.0-source
                                                  s.jar
-a----         6/16/2023  11:31 AM       11530020 iot-device-demo-ganerated-1.2.0-with-d
                                                  eps.jar
-a----         6/16/2023  11:31 AM           8031 iot-device-demo-ganerated-1.2.0.jar
```

2. Run the **java -jar .\target\iot-device-demo-ganerated-1.2.0-with-deps.jar ssl://*Domain name*:8883 *device_id secret*** command. The three parameters are the device access address, device ID, and password, respectively. Run the generated demo.

```
D:\git\huaweicloud-iot-device-sdk-java\generated-demo> java -jar .\target\iot-device-demo-ganerated-1.2.0-with-deps.jar ssl://Domain name:8883 5e06bfee334dd4f33759f5b3_demo secret
2024-04-17 15:50:53  INFO AbstractService:73 - create device, the deviceId is 5e06bfee334dd4f33759f5b3_demo
2024-04-17 15:50:54 INFO MqttConnection:204 - try to connect to ssl://Domain name: 8883
2024-04-17 15:50:55 INFO MqttConnection:228 - connect success, the uri is ssl://Domain name: 8883
2024-04-17 15:50:55  INFO MqttConnection:268 - publish message topic is $oc/devices/5e06bfee334dd4f33759f5b3_demo/sys/events/up, msg =
{"object_device_id":"5e06bfee334dd4f33759f5b3_demo","services":[{"paras":
```


{"type":"DEVICE_STATUS","content":"connect success","timestamp":"1713340255148"},"service_id":"$log","event_type":"log_report","event_time":"20240417T075055Z","event_id":null}]}
2024-04-17 15:50:55 INFO MqttConnection:111 - Mqtt client connected. address is ssl://***Domain name***: 8883
2024-04-17 15:50:55 INFO MqttConnection:268 - publish message topic is $oc/devices/***5e06bfee334dd4f33759f5b3_demo***/sys/events/up, msg = {"object_device_id":"***5e06bfee334dd4f33759f5b3_demo***","services":[{"paras":{"device_sdk_version":"JAVA_v1.2.0","fw_version":null,"sw_version":null},"service_id":"$sdk_info","event_type":"sdk_info_report","event_time":"20240417T075055Z","event_id":null}]}
2024-04-17 15:50:55 INFO MqttConnection:268 - publish message topic is $oc/devices/***5e06bfee334dd4f33759f5b3_demo*** /sys/events/up, msg = {"object_device_id":"***5e06bfee334dd4f33759f5b3_demo*** ","services": [{"paras":{"type":"DEVICE_STATUS","content":"connect complete, the url is ssl://***Domain name*** :8883","timestamp":"1713340255496"},"service_id":"$log","event_type":"log_report","event_time":"20240417T075055Z","event_id":null}]}
2024-04-17 15:51:03 INFO smokeDetectorService:78 - report property alarm value = 50
2024-04-17 15:51:03 INFO smokeDetectorService:104 - report property temperature value = 0.3648571367849047
2024-04-17 15:51:03 INFO smokeDetectorService:91 - report property smokeConcentration value = 0.679772877336927
2024-04-17 15:51:03 INFO smokeDetectorService:117 - report property humidity value = 15
2024-04-17 15:51:03 INFO MqttConnection:268 - publish message topic is $oc/devices/***5e06bfee334dd4f33759f5b3_demo***/sys/properties/report, msg = {"services":[{"properties":{"alarm":50,"temperature":0.3648571367849047,"smokeConcentration":0.679772877336927,"humidity":15},"service_id":"smokeDetector","event_time":"20240417T075103Z"}]}

To modify the extended code, proceed as follows:

Service definition and registration have already been completed through the generated code. You only need to make small changes to the code.

1. Command API: Add specific implementation logic.

```
**********  ommands ***********/
@DeviceCommand
public CommandRsp ringAlarm (Map<String, Object> paras) {
  //todo Add command processing code here.
  return new CommandRsp(0);
}
```

2. **getter** method: Change the value return mode of the generated code from returning a random value to reading from the sensor.
3. **setter** method: Add specific processing logic, such as delivering instructions to the sensor, because the generated code only modifies and saves the properties.

Developing a Gateway

Gateways are special devices that provide child device management and message forwarding in addition to the functions of common devices. The SDK provides the **AbstractGateway** class to simplify gateway implementation. This class can collect and save child device information (with a data persistence API), forward message responses (with a message forwarding API), and report child device list, properties, statuses, and messages.

- **AbstractGateway Class**

  Inherit this class to provide APIs for persistently storing device information and forwarding messages to child devices in the constructor.


```
public abstract void onSubdevCommand(String requestId, Command command);

public abstract void onSubdevPropertiesSet(String requestId, PropsSet propsSet);

public abstract void onSubdevPropertiesGet(String requestId, PropsGet propsGet);

public abstract void onSubdevMessage(DeviceMessage message);
```

- **iot-gateway-demo Code**

  The **iot-gateway-demo** project implements a simple gateway with **AbstractGateway** to connect TCP devices. The key classes include:

  SimpleGateway: inherited from **AbstractGateway** to manage child devices and forward messages to child devices.

  StringTcpServer: implements a TCP server based on Netty. In this example, child devices support the TCP protocol, and the first message is used for authentication.

  SubDevicesFilePersistence: persistently stores child device information in a JSON file and caches the file in the memory.

  Session: stores the mapping between device IDs and TCP channels.

- **SimpleGateway Class**

  Adding or Deleting a Child Device

  Adding a child device: **onAddSubDevices** of **AbstractGateway** can store child device information. Additional processing is not required, and **onAddSubDevices** does not need to be overridden for **SimpleGateway**.

  Deleting a child device: You need to modify persistently stored information of the child device and disconnect the device from the platform. Therefore, **onDeleteSubDevices** is overridden to add the link release logic, and **onDeleteSubDevices** in the parent class is called.

```
@Override
  public int onDeleteSubDevices(SubDevicesInfo subDevicesInfo) {

      for (DeviceInfo subdevice : subDevicesInfo.getDevices()) {
          Session session = nodeIdToSesseionMap.get(subdevice.getNodeId());
          if (session != null) {
              if (session.getChannel() != null) {
                  session.getChannel().close();
                  channelIdToSessionMap.remove(session.getChannel().id().asLongText());
                  nodeIdToSesseionMap.remove(session.getNodeId());
              }
          }
      }
      return super.onDeleteSubDevices(subDevicesInfo);

  }
```

- **Processing Messages to Child Devices**

  The gateway needs to forward messages received from the platform to child devices. The messages from the platform include device messages, property reading/writing, and commands.

  – **Device messages**: Obtain the nodeId based on the deviceId, and then obtain the session of the device to get a channel for sending messages. You can choose whether to convert messages during forwarding.

```
@Override
  public void onSubdevMessage(DeviceMessage message) {

      // Each platform API carries a deviceId, which consists of a nodeId and productId.
```



```
        //deviceId = productId_nodeId
        String nodeId = IotUtil.getNodeIdFromDeviceId(message.getDeviceId());
        if (nodeId == null) {
            return;
        }

        // Obtain the session based on the nodeId for a channel.
        Session session = nodeIdToSesseionMap.get(nodeId);
        if (session == null) {
            log.error("subdev is not connected " + nodeId);
            return;
        }
        if (session.getChannel() == null){
            log.error("channel is null " + nodeId);
            return;
        }

        // Directly forward messages to the child device.
        session.getChannel().writeAndFlush(message.getContent());
        log.info("writeAndFlush " + message);
    }
```

- **Property Reading and Writing**

  Property reading and writing include property setting and query.

  Property setting:

```
@Override
    public void onSubdevPropertiesSet(String requestId, PropsSet propsSet) {

        if (propsSet.getDeviceId() == null) {
            return;
        }

        String nodeId = IotUtil.getNodeIdFromDeviceId(propsSet.getDeviceId());
        if (nodeId == null) {
            return;
        }

        Session session = nodeIdToSesseionMap.get(nodeId);
        if (session == null) {
            return;
        }

        // Convert the object into a string and send the string to the child device. Encoding/
Decoding may be required in actual situations.
        session.getChannel().writeAndFlush(JsonUtil.convertObject2String(propsSet));

        // Directly send a response. A more reasonable method is to send a response after the
child device processes the request.
        getClient().respondPropsSet(requestId, IotResult.SUCCESS);

        log.info("writeAndFlush " + propsSet);

    }
```

  Property query:

```
@Override
    public void onSubdevPropertiesGet(String requestId, PropsGet propsGet) {

        // Send a failure response. It is not recommended that the platform directly reads the
properties of the child device.
        log.error("not supporte onSubdevPropertiesGet");
        deviceClient.respondPropsSet(requestId, IotResult.FAIL);
    }
```

- **Commands**: The procedure is similar to that of message processing. Different types of encoding/decoding may be required in actual situations.


```
@Override
    public void onSubdevCommand(String requestId, Command command) {

        if (command.getDeviceId() == null) {
            return;
        }

        String nodeId = IotUtil.getNodeIdFromDeviceId(command.getDeviceId());
        if (nodeId == null) {
            return;
        }

        Session session = nodeIdToSesseionMap.get(nodeId);
        if (session == null) {
            return;
        }

        // Convert the command object into a string and send the string to the child device.
Encoding/Decoding may be required in actual situations.
        session.getChannel().writeAndFlush(JsonUtil.convertObject2String(command));

        // Directly send a response. A more reasonable method is to send a response after the
child device processes the request.
        getClient().respondCommand(requestId, new CommandRsp(0));
        log.info("writeAndFlush " + command);
    }
```

- **Upstream Message Processing**

  Upstream message processing is implemented by **channelRead0** of **StringTcpServer**. If no session exists, create a session.

  If the child device information does not exist, the session cannot be created and the connection is rejected.

```
@Override
    protected void channelRead0(ChannelHandlerContext ctx, String s) throws Exception {
        Channel incoming = ctx.channel();
        log.info("channelRead0" + incoming.remoteAddress() + " msg :" + s);

        // Create a session for the first message.
// Create a session for the first message.
        Session session = simpleGateway.getSessionByChannel(incoming.id().asLongText());
        if (session == null) {
            String nodeId = s;
            session = simpleGateway.createSession(nodeId, incoming);

            // The session fails to create and the connection is rejected.
            if (session == null) {
                log.info("close channel");
                ctx.close();
            }
        }
```

  If the session exists, the message is forwarded.

```
else {
            // Call reportSubDeviceProperties to report properties of the child device.
            DeviceMessage deviceMessage = new DeviceMessage(s);
            deviceMessage.setDeviceId(session.getDeviceId());
            simpleGateway.reportSubDeviceMessage(deviceMessage, null);

        }
```

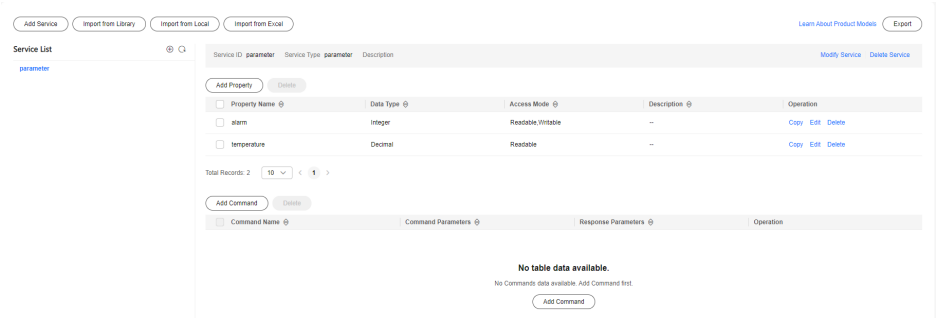
  For details about the gateway, view the source code. The demo is open-source and can be extended as required. For example, you can modify the persistence mode, add message format conversion during forwarding, and support other device access protocols.

- **Using iot-gateway-demo**

  a. Create a product for the child device. For details, see **Creating a Product**.

  b. Define a model in the created product and add a service whose ID is **parameter**. Add **alarm** and **temperature** properties, as shown in the following figure.

  Figure 4-9 Model definition - Child device product

  c. Modify the **main** function of **StringTcpServer** by replacing the constructor parameters, and run this class.

  ```
  simpleGateway = new SimpleGateway(new SubDevicesFilePersistence(),
                  "ssl://iot-acc.cn-north-4.myhuaweicloud.com:8883",
                  "5e06bfee334dd4f33759f5b3_demo", "mysecret");
  ```

  d. After the gateway is displayed as **Online** on the platform, add a child device.

  Figure 4-10 Device - Adding a child device

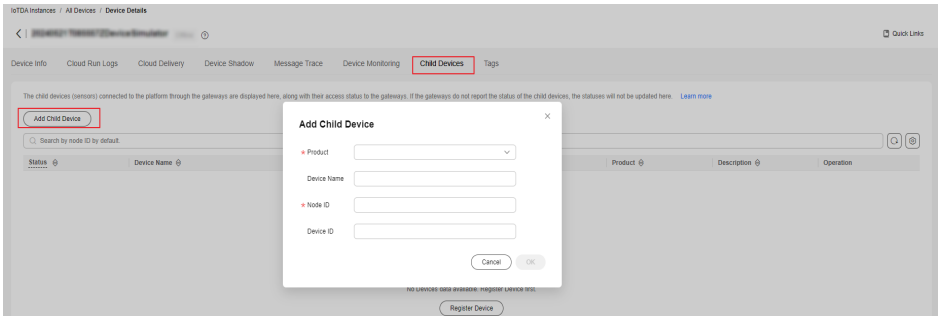
  

  Table 4-2 Child device parameters

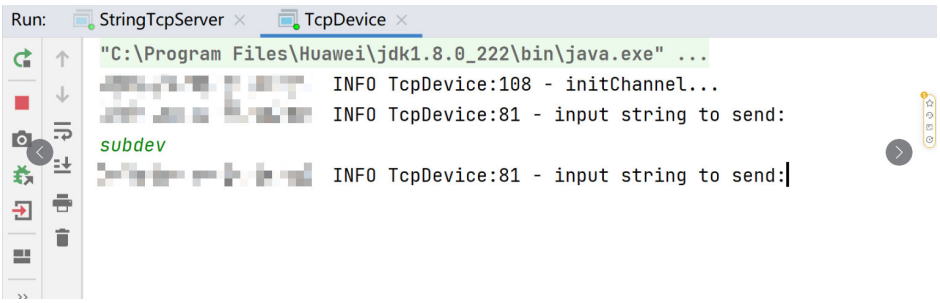
Parameter	Description
Product	Product to which the child device belongs. Select the product created in 1.
Device Name	Customize a device name, for example, **subdev_name**.
Node ID	Enter **subdev**.
Device ID	This parameter is optional and is automatically generated.

A log similar to the following is displayed on the gateway:

2024-04-16 21:00:01 INFO SubDevicesFilePersistence:112 - add subdev, the nodeId is subdev

e. Run the TcpDevice class. After the connection is set up, enter the node ID of the child device registered in step 3, for example, **subdev**.

Figure 4-11 Child device connection

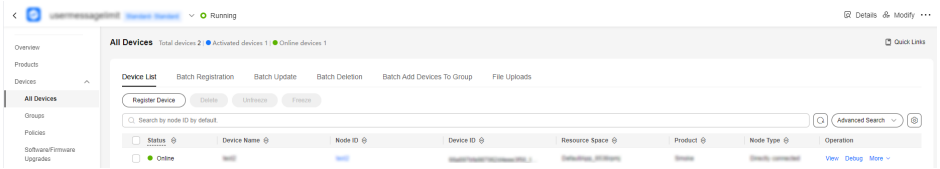


A log similar to the following is displayed on the gateway:

2024-04-16 21:00:54 INFO StringTcpServer:196 - initChannel: /127.0.0.1:21889
2024-04-16 21:01:00 INFO StringTcpServer:137 - channelRead0 is /127.0.0.1:21889, the msg is ***subdev***
2024-04-16 21:01:00 INFO SimpleGateway:100 - create new session ok, the session is Session{nodeId='***subdev***', channel=[id: 0xf9b89f78, L:/127.0.0.1:8080 - R:/127.0.0.1:21889], deviceId='***subdev_deviceId***'}

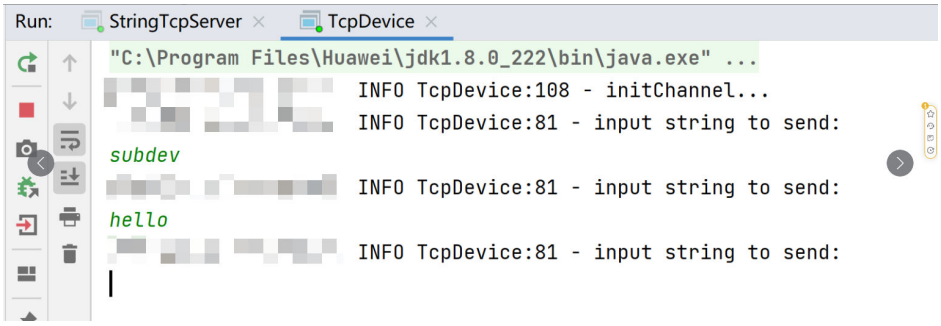
f. Check whether the child device is online on the platform.

Figure 4-12 Device list - Device online status



g. Enable the child device to report messages.

Figure 4-13 Enable the child device to report messages.



Logs similar to the following show that the message is reported.

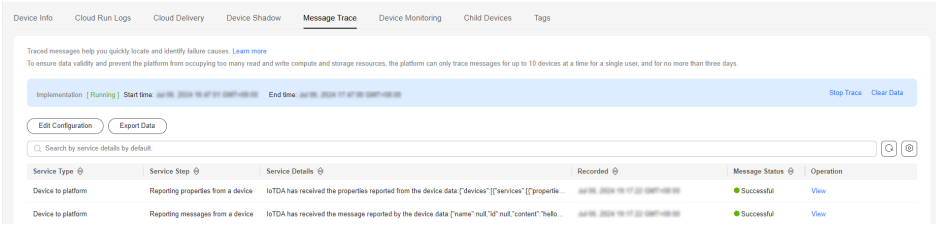
2024-04-16 21:02:36 INFO StringTcpServer:137 - channelRead0 is /127.0.0.1:***21889***, the msg is hello
2024-04-16 21:02:36 INFO MqttConnection:299 - publish message topic is $oc/devices/***5e06bfee334dd4f33759f5b3_demo***/sys/messages/up, msg = {"name":null,"id":null,"content":"hello","object_device_id":"***subdev_deviceId***"}


```
2024-04-16 21:02:36  INFO MqttConnection:299 - publish message topic is $oc/devices/
5e06bfee334dd4f33759f5b3_demo/sys/gateway/sub_devices/properties/report, msg =
{"devices":[{"services":[{"properties":
{"temprature":2,"alarm":1},"service_id":"parameter","event_time":null}],"device_id":"subdev_devi
ceId"}]]
```

h. View the messages traced.

Click **Message Trace** on the gateway details page. Send data from the child device to the platform, and view the messages after a while.

Figure 4-14 Message tracing - Directly connected device



4.2.3 IoT Device SDK (C)

The IoT Device SDK (C) provides abundant demo code for devices to communicate with the platform and implement device, gateway, and Over-The-Air (OTA) services. For details on the integration guide, see **IoT Device SDK (C) Development Guide**.

Requirements

- The SDK runs on Linux.
- The SDK depends on the OpenSSL and Paho libraries. If you have your own compilation chain, compile library files such as OpenSSL, Paho, zlib, and Huawei secure function library.
- For some devices that are connected in MCU+module mode, use the C Tiny SDK for development.

NOTE

For details, see **README**.

Change History

Table 4-3 Change history

Version	Change	Description
1.2.0	Function enhancement	Added the SDK test code and demo, and optimized the code usage.
1.1.5	Function enhancement	Updated the OTA upgrade transmission format.
1.1.4	Function enhancement	Fixed the issue of remote login packet reporting timeout.

Version	Change	Description
1.1.3	Function enhancement	Updated the conf\rootcert.pem certificates.
1.1.2	New function	Added device rules, M2M, GN compilation file, anomaly detection, timestamp printed in logs, MQTT_DEBUG, Chinese cryptographic algorithm, remote configuration, and device-cloud secure communication (soft bus).
1.1.1	New function	Added SSH remote O&M.
1.1.0	New function	Supported MQTT 5.0. Optimized the cod to resolve the memory overflow issue.
1.0.1	Function enhancement	Added application scenarios, where MQTTS does not verify the platform public key, using TLS version is V1.2, and adding message storage examples.
0.9.0	New function	Added the API for the gateway to update the child device status.
0.8.0	Function enhancement	Added the access domain name (iot-mqtts.cn-north-4.myhuaweicloud.com) and root certificates. If the device uses the old domain name (iot-acc.cn-north-4.myhuaweicloud.com) for access, use the v0.5.0 SDK.
0.5.0	Function enhancement	Preset the device access address and the matching CA certificate in the SDK to support interconnection with the Huawei Cloud IoT platform.

4.2.4 IoT Device SDK (C#)

The IoT Device SDK (C#) provides abundant demo code for devices to communicate with the platform and implement advanced services such as device, gateway, and Over-The-Air (OTA) services. For details about the integration guide, see **IoT Device SDK (C#) Development Guide**.

Requirements

- DotNet SDK 8.0 has been installed.
  - **.NET installation guide**
  - **.NET 8.0.**
- The corresponding IDE (Visual Studio Code 2017+, Rider 17.0.6+) has been installed. This SDK does not depend on the IDE. You can select the IDE or directly use the CLI as required.

NOTE

For details, see **README**.

Change History

Table 4-4 Change history

Version	Change	Description
1.3.4	Function enhancement	1. Optimized the log printing function. 2. Modified the topic returned by SubscribeTopic starting with oc. 3. Optimized demos. 4. Fixed the bug of the gateway interface. 5. Upgraded the target framework. 6. Optimized other features.
1.3.3	New function	Supported gateway mode for OTA upgrade.
1.3.2	Function enhancement	Updated the CA certificate for the server.
1.3.1	Fixing	Resolved issues such as null pointer exceptions and MQTT object release failures.
1.3.0	New function	Supported OBS-based upgrade of software and firmware packages.
1.2.0	New function	Added the generic-protocol function.
1.1.1	Function enhancement	Added the function of deleting child devices from a gateway and optimized the description.
1.1.0	New function	Added the gateway and product model functions.
1.0.0	First release	Provided basic device access capabilities. Preset the device access address and the CA certificate matching Huawei IoTDA in the SDK.

4.2.5 IoT Device SDK (Android)

The IoT Device SDK (Android) provides abundant demo code for devices to communicate with the platform and implement advanced services such as device, gateway, and Over-The-Air (OTA) services. For details on the integration guide, see **IoT Device SDK (Android) Development Guide**.

Requirements

- Android Studio has been installed.

NOTE

For details, see **README**.

Change History

Table 4-5 Change history

Version	Change	Description
1.0.0	First release	Provided basic device access capabilities.

4.2.6 IoT Device SDK (Go)

The IoT Device SDK (Go) provides abundant demo code for devices to communicate with the platform and implement advanced services such as device and Over-The-Air (OTA) services. For details on the integration guide, see **IoT Device SDK (Go) Development Guide**.

Requirements

- Go 3.18 has been installed.
- The dependencies have been installed based on go.mod.

NOTE

For details, see **README**.

Change History

Table 4-6 Change history

Version	Change	Description
v1.0.0	New function	Provided capabilities for connections to the Huawei Cloud IoT platform to facilitate service scenarios such as secure access, device management, data collection, command delivery, device provisioning, and device rules.

4.2.7 IoT Device SDK Tiny (C)

The IoT Device SDK Tiny is lightweight interconnection middleware deployed on devices that have WAN capabilities and limited power consumption, storage, and computing resources. After the IoT Device SDK Tiny is deployed on such devices, you only need to call APIs to enable the devices to connect to the IoT platform,

report data, and receive commands. For details about the integration, see **development guide on device-cloud communication components**.

NOTE

The IoT Device SDK Tiny can run on devices that do not run Linux OS, and can also be integrated into modules. However, it does not provide gateway services.

Requirements

- It adapts to LiteOS, Linux, macOS, and FreeRTOS. You can modify the SDK to **adapt to other environments**.
- For details about different modules, see the **SDK development board porting list**.

4.2.8 IoT Device SDK (Python)

The IoT Device SDK (Python) provides abundant demo code for devices to communicate with the platform and implement device, gateway, and Over-The-Air (OTA) services. For details, see **IoT Device SDK (Python) Development Guide**.

Requirements

- Python 3.11.4 has been installed.
- The third-party class library paho-mqtt 2.0.0 has been installed (mandatory).
- The third-party class library schedule 1.2.2 has been installed (mandatory).
- The third-party class library apscheduler 3.10.4 has been installed (mandatory).
- The third-party class library requests 2.32.2 has been installed (optional, used in the demo of gateway and child device management).
- The third-party class library tornado 6.3.3 has been installed (optional, used in the demo of gateway and child device management).

NOTE

For details about how to install the components, see **IoT Device SDK (Python) Usage Guide**.

Change History

Table 4-7 Change history

Version	Change	Description
1.2.0	New function	Added the functions of rule engine, device provisioning, customized reconnection upon disconnection, and component version upgrade.
1.1.4	New function	Supported gateway mode for OTA upgrade.

Version	Change	Description
1.1.3	Function enhancement	Updated the CA certificate for the server.
1.1.2	New function	Supported micropython and the corresponding demo, OTA downloading from OBS, and description documents.
1.1.1	New function	Provided capabilities for connections to the Huawei Cloud IoT platform to facilitate service scenarios such as secure access, device management, data collection, and command delivery.

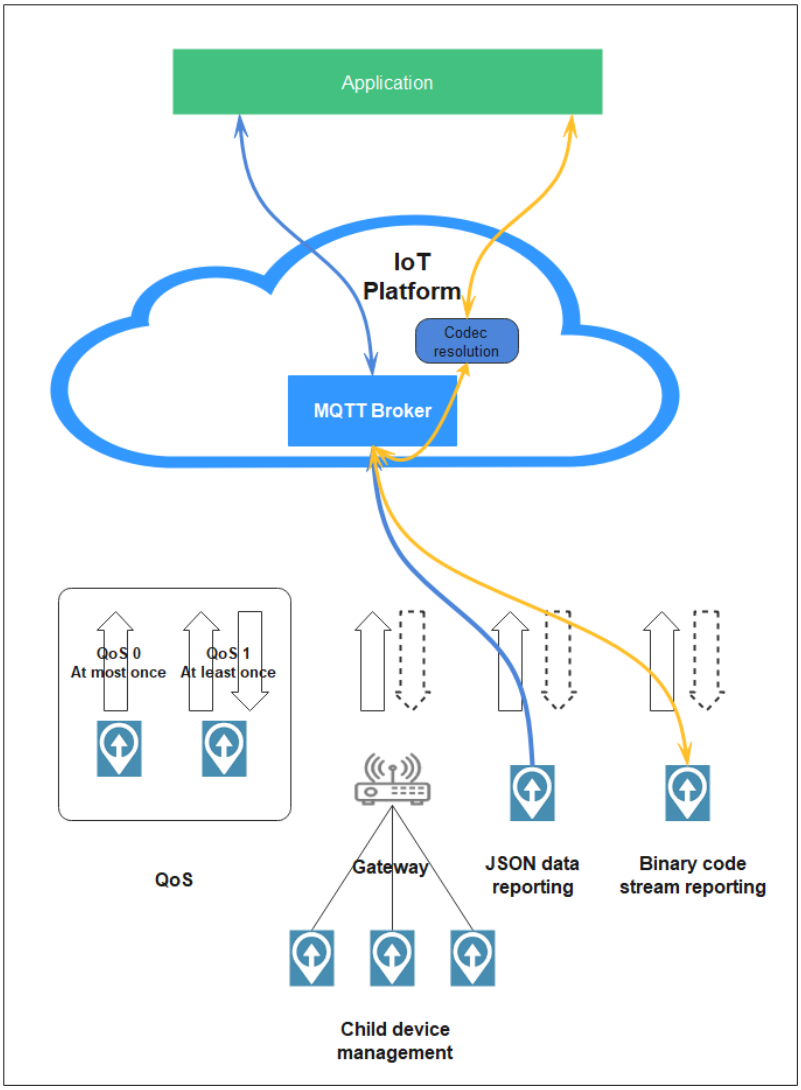
4.3 Using MQTT Demos for Access

4.3.1 MQTT Usage Guide

Overview

Message Queuing Telemetry Transport (MQTT) is a publish/subscribe messaging protocol that transports messages between clients and a server. 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 connections. To learn more about the MQTT syntax and interfaces, click **here**.

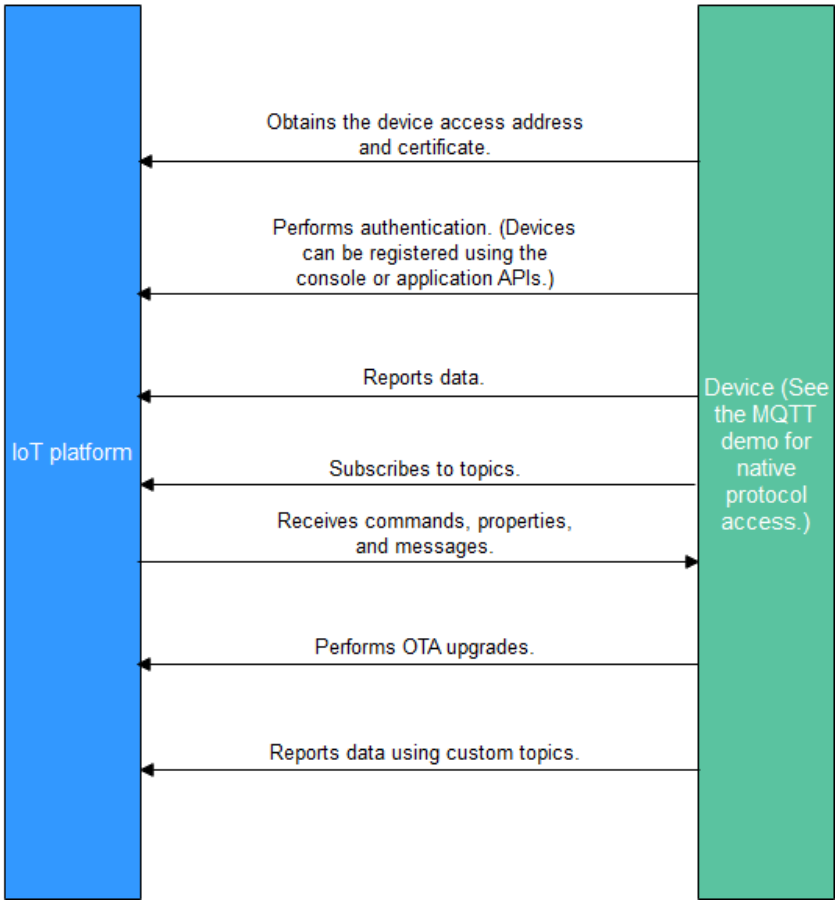
MQTTS is a variant of MQTT that uses TLS encryption. MQTTS devices communicate with the platform using encrypted data transmission.



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

Item	Constraint
Supported MQTT version	MQTT v3.1, v3.1.1, and v5.0 are supported. QoS 2, and will and retained messages are not supported.

Item	Constraint
Differences from the standard MQTT protocol	• QoS 0 and QoS 1 are supported. • Custom topics are supported. • QoS 2 is not supported. • **will** and **retain msg** are not supported.
Security level supported by MQTTS	TCP channel + TLS (TLS 1.3 or earlier)
Maximum number of MQTT connection requests allowed for an account per second	No limit
Maximum number of MQTT connections allowed for a device per minute	1
Maximum throughput of an MQTT connection per second, including directly connected devices and gateways	3 KB/s
Maximum length of a message reported by an MQTT device (A message with the length greater than this value is rejected.)	1 MB
Recommended heartbeat interval for MQTT connections	Range: 30s to 1200s; recommended: 120s
Custom topic	Supported
Publish/Subscribe	A device can only publish and subscribe to messages of its own topics.
Maximum number of subscriptions per subscription request	No limit

NOTE

You are advised to use encrypted channels (port 8883) for secure communications between devices and the platform.

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.
	Batch reporting device properties	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 1.0, TLS 1.1, TLS 1.2, and TLS 1.3 are supported. TLS 1.0 and TLS 1.1 will soon be deprecated. Therefore, TLS 1.2 and TLS 1.3 are recommended. The platform only supports the following cipher suites for TLS connections:

- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA

- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384

4.3.2 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**.

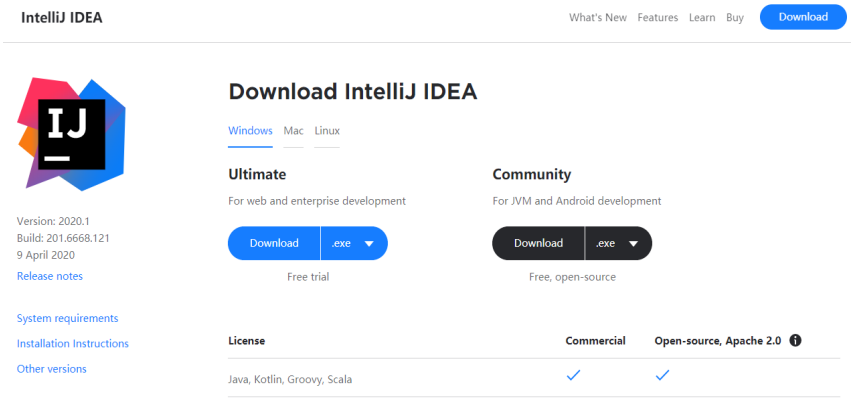
Prerequisites

- 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 64-bit IntelliJ IDEA 2019.2.3 Ultimate as an example.

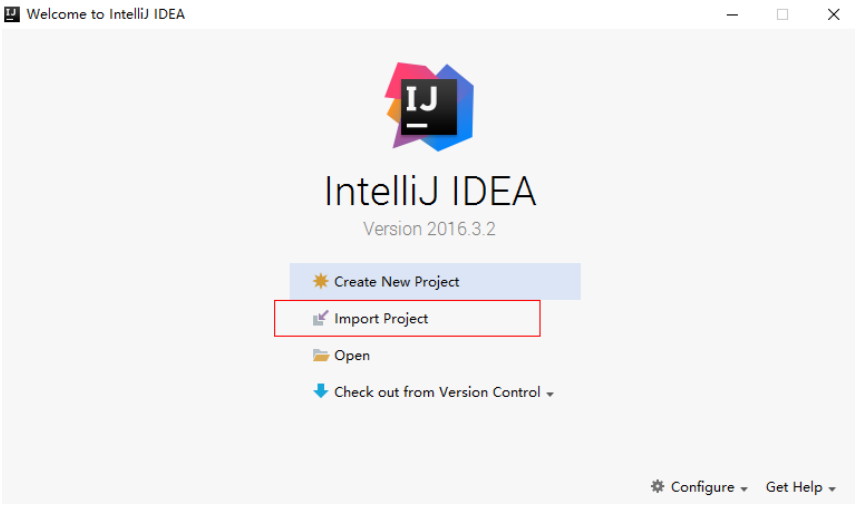


2. After the download is complete, run the installation file and install IntelliJ IDEA as prompted.

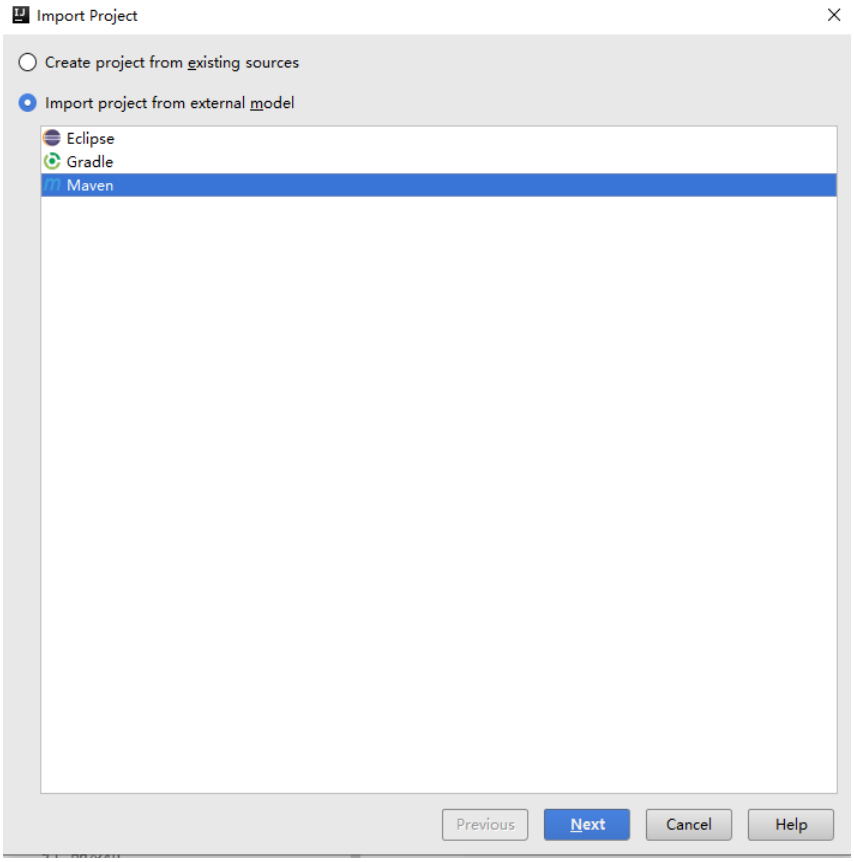
Importing Sample Code

Step 1 Download the **Java demo**.

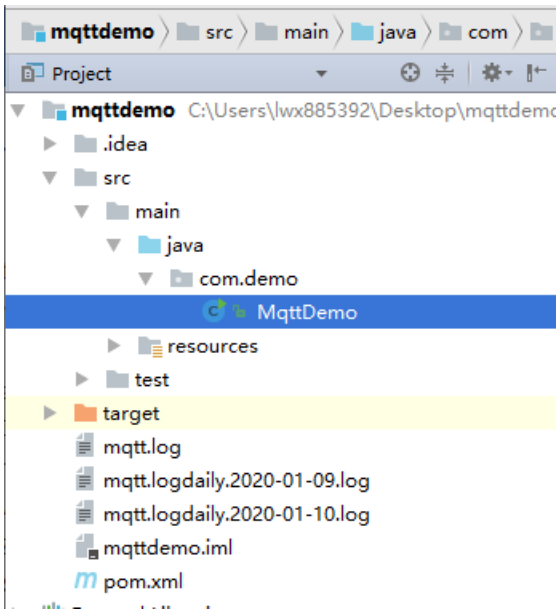
Step 2 Open the IDEA developer tool and click **Import Project**.



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.

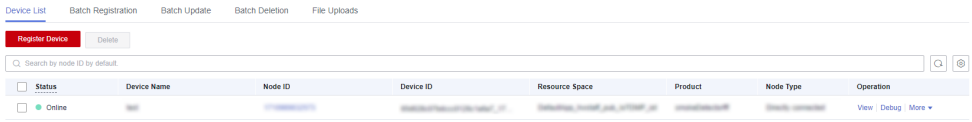
```
client.connect(options, null, new IMqttActionListener()
```

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 4-15 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.

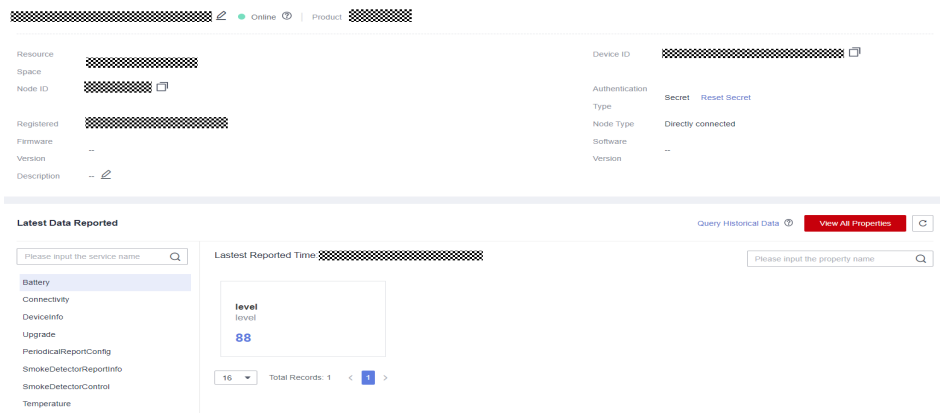
After a device or gateway is connected to the platform, you can call **MqttClient.publish(String topic,MqttMessage message)** to report device properties to the platform.

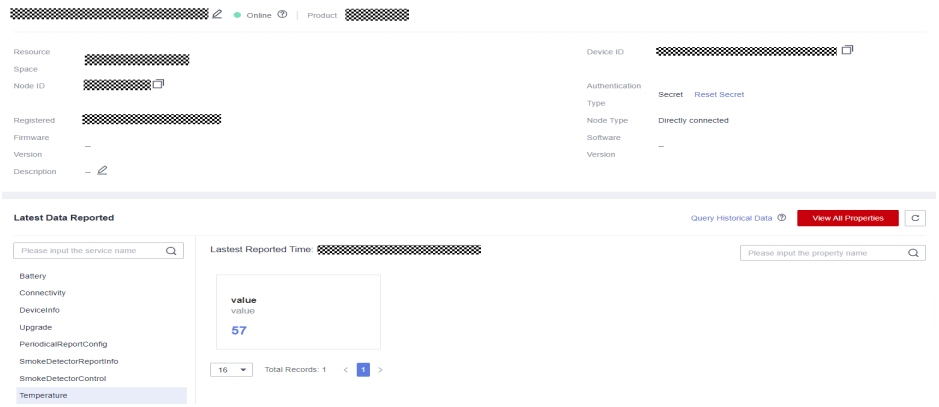
getRreportTopic() is used to obtain the topic for reporting data.

```
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**.





NOTE

If no latest data is displayed on the device details page, modify the services and properties in the product model to ensure that the reported services and properties are the same as those defined in the product model. Alternatively, go to the **Products** > **Model Definition** page and delete all services.

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**.

4.3.3 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**.

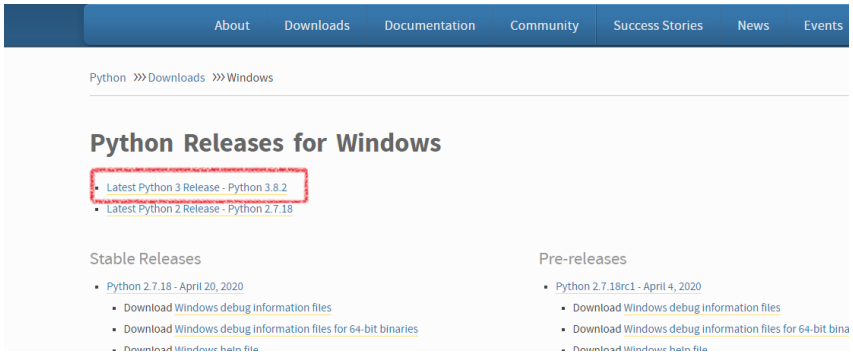
Prerequisites

- 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**.

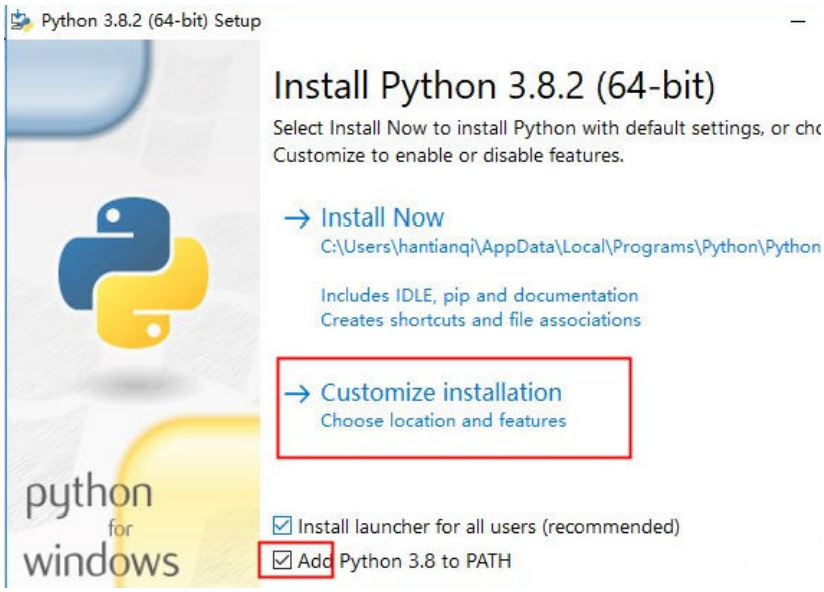
- You have created a product and a device on the **IoTDA console**. For details, see **Creating a Product**, **Registering an Individual Device**, and **Registering a Batch of Devices**.

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.)



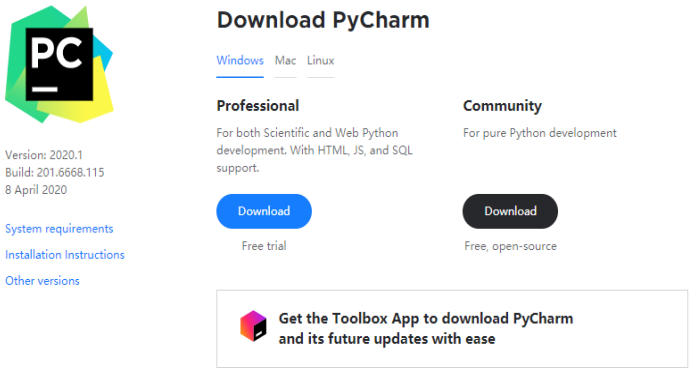
  b. After the download is complete, run the .exe file to install Python.
  c. Select **Add python 3.8 to PATH** (if it is not selected, you need to manually configure environment variables), click **Customize installation**, and install Python as prompted.



  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**.

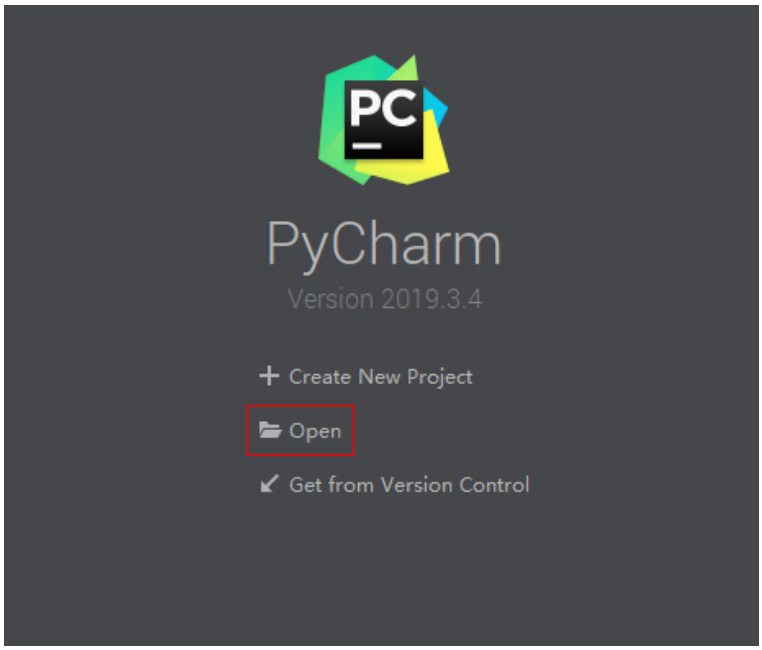


  The professional edition is recommended.
  b. Run the .exe file and install PyCharm as prompted.

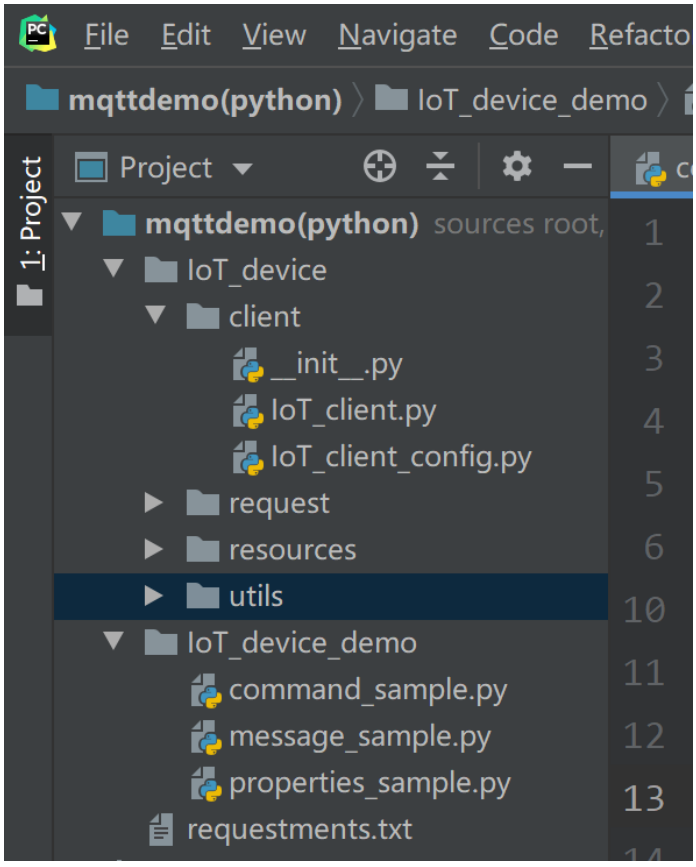
Importing Sample Code

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.

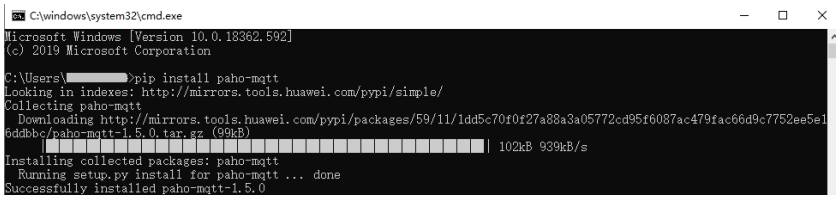
  DigiCertGlobalRootCA.crt.pem is used by the device to verify the platform identity when the device connects to the platform. You can download the certificate file using the link provided in **Certificates**.

- **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:

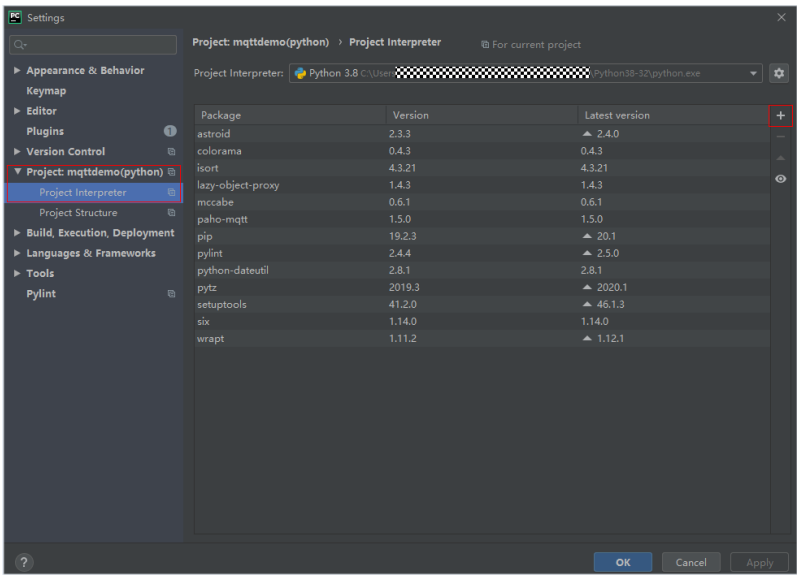
- Method 1: Use the pip tool to install paho-mqtt in the CLI. (The tool is already provided when installing Python.)

  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.

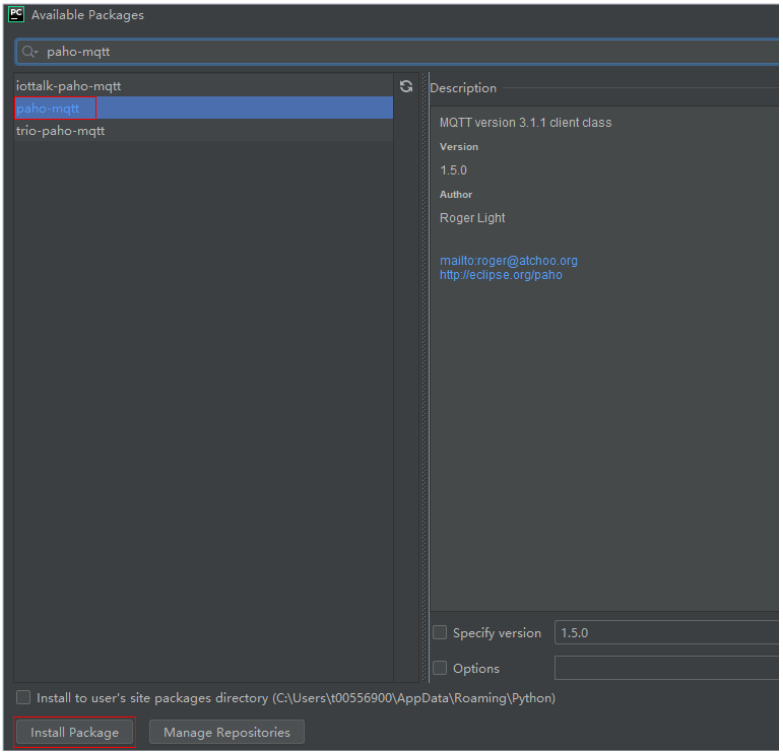


- Method 2: Install paho-mqtt using PyCharm.

  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

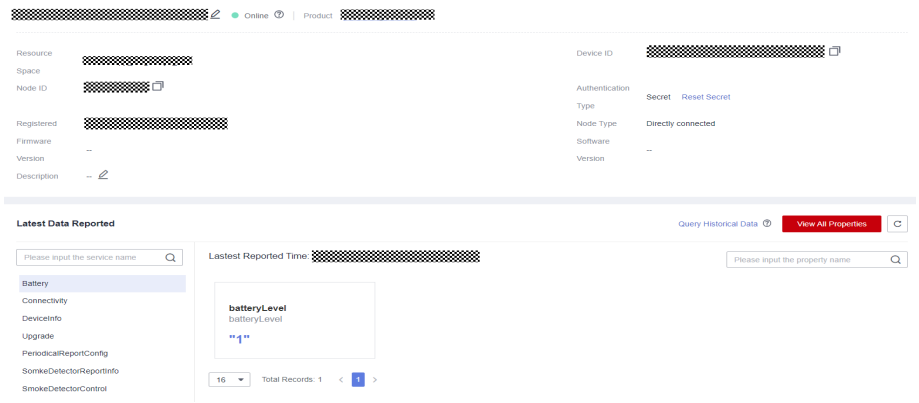
Devices can report their properties to the platform. For details about the API, see **Device Reporting Properties**.

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.



NOTE

If no latest data is displayed on the device details page, modify the services and properties in the product model to ensure that the reported services and properties are the same as those defined in the product model. Alternatively, go to the **Products** > **Model Definition** page and delete all services.

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**.

4.3.4 Android Demo Usage Guide

Overview

This topic usesAndroid 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

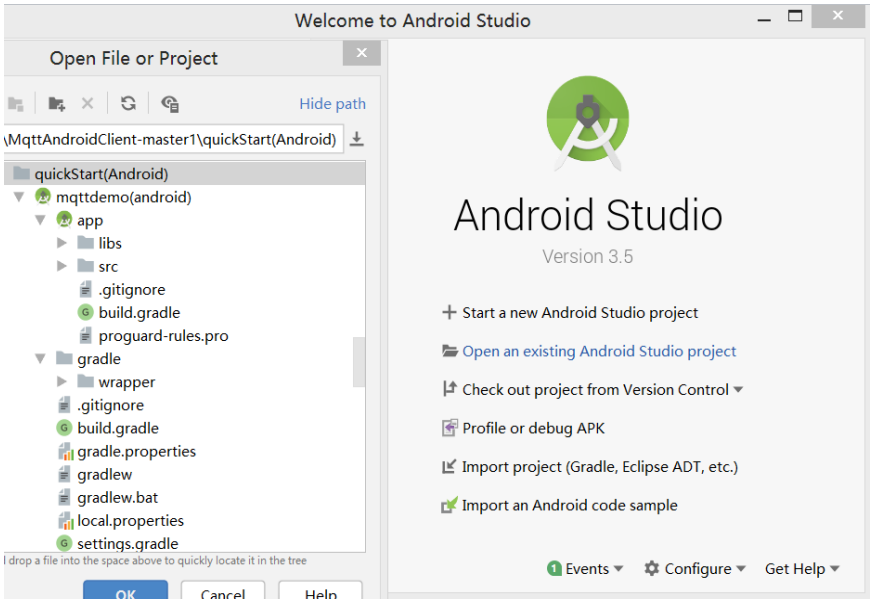
Platform	Android Studio package	Size	SHA-256 checksum
Windows (64-bit)	android-studio-ide-192.6392135-windows.exe Recommended	756 MB	07b6df807fda59e69f05b85ff6f6bd0c70d09e57fb151197155ef5f115f96e59
	android-studio-ide-192.6392135-windows.zip No .exe installer	770 MB	24f8f9ce467b935c25d89b90cad402d21dd45d4ba9af1ad35baeeb414609e483
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.

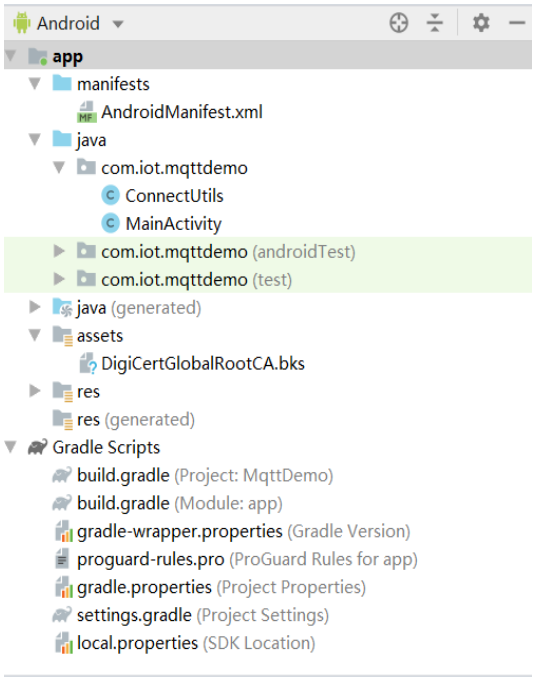
Importing Sample Code

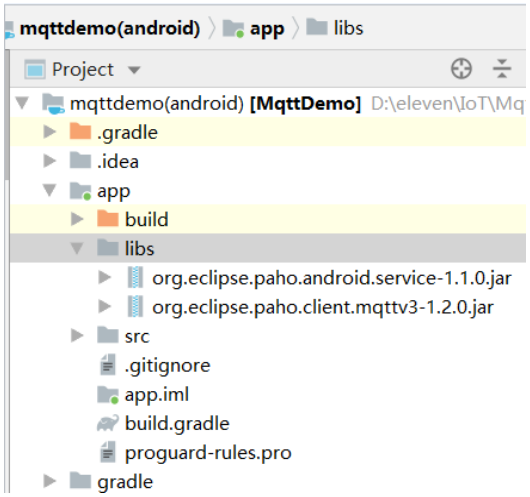
Step 1 Download the sample code **quickStart(Android)**.

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.

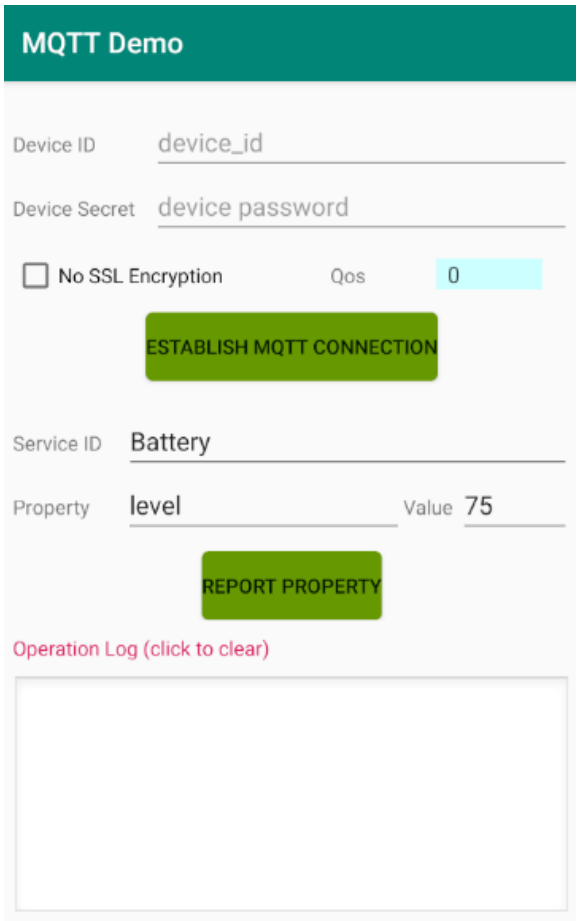
```
<service android:name="org.eclipse.paho.android.service.MqttService" />
```

- **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, QoS2 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_connect_ssl.setText ("SSL encryption");
           } else {
               isSSL = false;
               checkbox_mqtt_connect_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**.

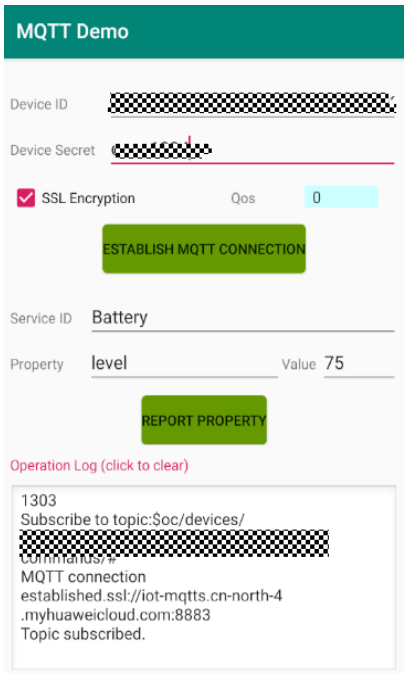
The **MainActivity** class provides the methods for delivering subscription commands to topics, subscribing to topics, and unsubscribing from topics.

```
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:



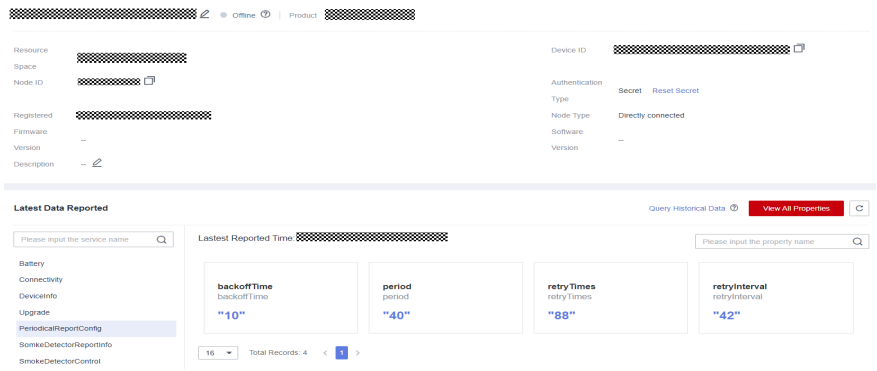
Reporting Properties

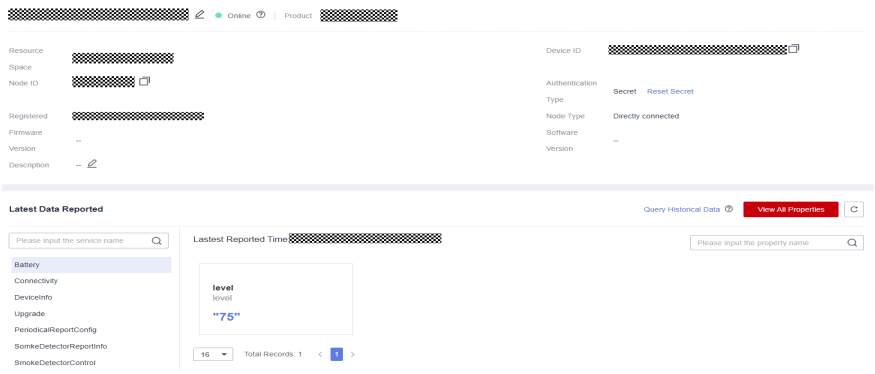
Devices can report their properties to the platform. For details about the API, see **Device Reporting Properties**.

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.





NOTE

If no latest data is displayed on the device details page, modify the services and properties in the product model to ensure that the reported services and properties are the same as those defined in the product model. Alternatively, go to the **Products** > **Model Definition** page and delete all services.

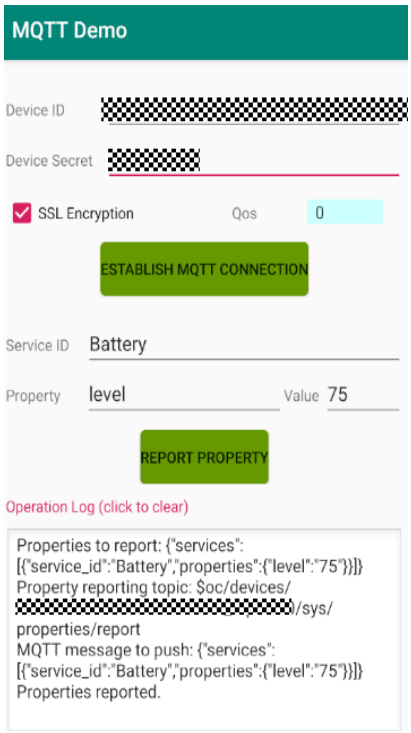
Receiving a Command

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:



4.3.5 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**.

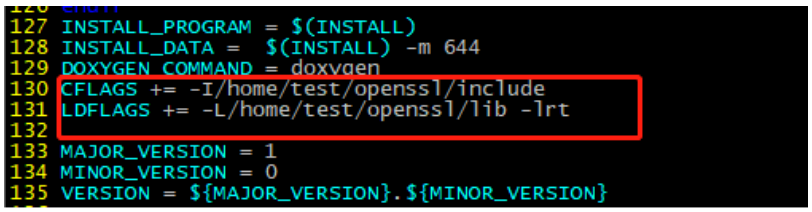
b. Decompress the package and upload it to the Linux compiler.

c. Modify the **makefile**.

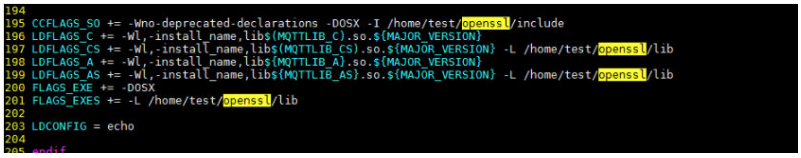
i. Run the following command to edit the **makefile**:
```
vim Makefile
```

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
```



iv. Replace the OpenSSL addresses of **CCDLAGS_SO**, **LDFLAGS_CS**, **LDFLAGS_AS** and **FLAGS_EXES** to the actual ones.

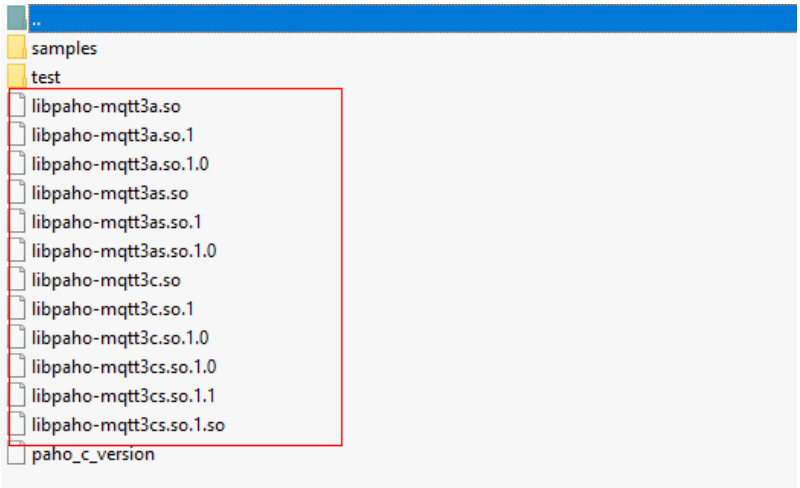


d. Start the compilation.

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.

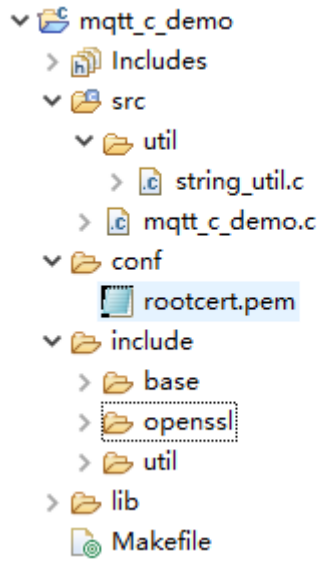
Currently, only **libpaho-mqtt3as** is used in the SDK. Copy the **libpaho-mqtt3as.so** and **libpaho-mqtt3as.so.1** files to the **lib** folder of the demo. Go back to the Paho source code directory, and copy **MQTTAsync.h**, **MQTTClient.h**, **MQTTClientPersistence.h**, **MQTTProperties.h**, **MQTTReasonCodes.h**, and **MQTTSubscribeOpts.h** in the **src** directory to the **include/base** directory of the demo.

> **CAUTION**
>
> 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 running 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 = "5ebac693352cfb02c567ec88_test2345"; //deviceId
char *password = "602d6cc77d87271be8f462f52d27d818";
```

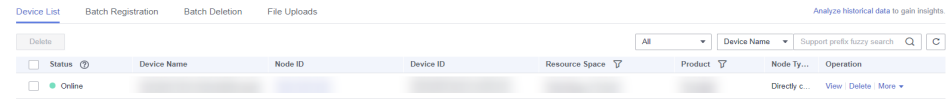
   Note: MQTTS uses port 8883 for access. If MQTT is used for access, the URL is **tcp://Domain name space:1883** and the port is 1883. For details about how to obtain the domain name space, see **Platform Connection Information**. The default heartbeat interval is 120 seconds. To change it, modify the **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.
```



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.

```
//subcribe
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

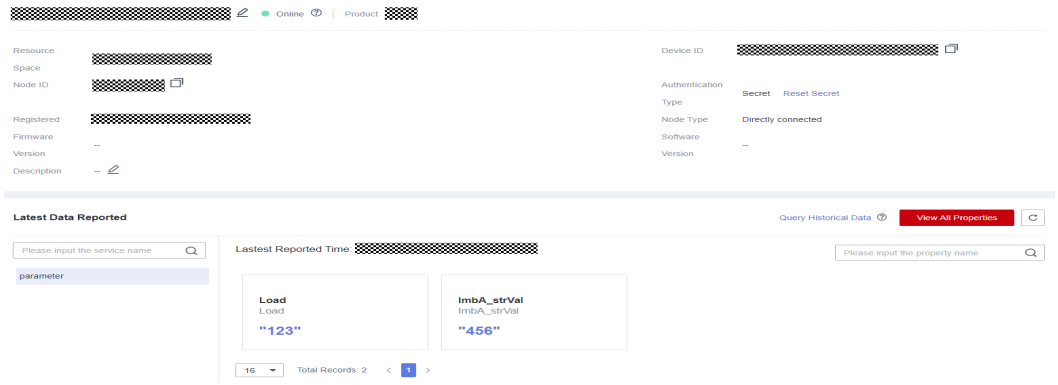
Devices can report their properties to the platform. For details, see **Reporting Device Properties**.

```
//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 4-16 Device details page



NOTE

If no latest data is displayed on the device details page, modify the services and properties in the product model to ensure that the reported services and properties are the same as those defined in the product model. Alternatively, go to the **Products** > **Model Definition** page and delete all services.

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; //can not return 0 here, otherwise the message won't update or something wrong would happen
}
```

NOTE

Synchronous commands require device responses. For details, see **Upstream Response Parameters**.

4.3.6 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**.

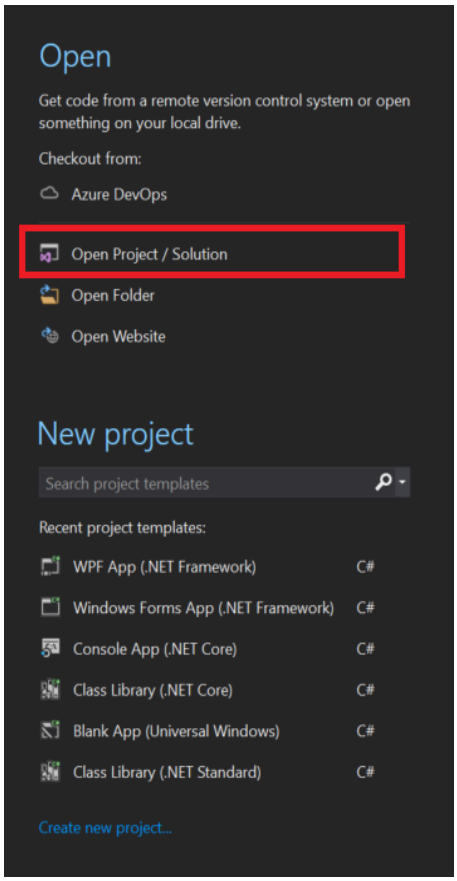
Preparations

- 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.

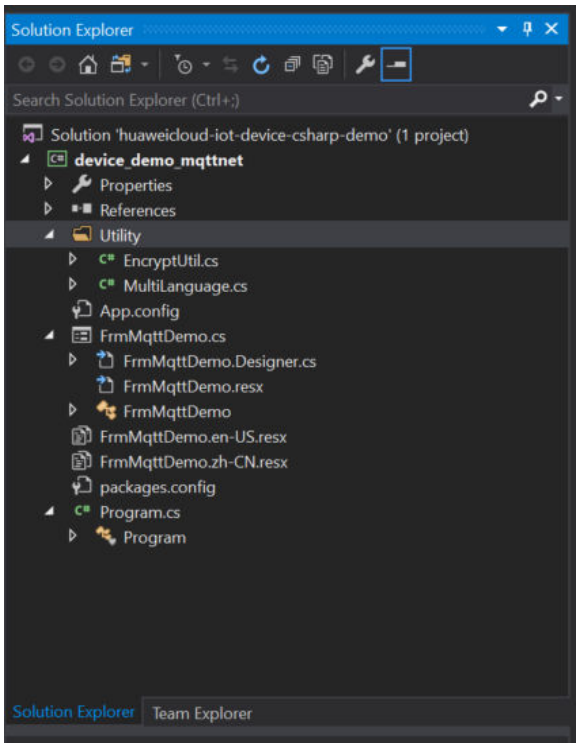
Importing Sample Code

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 key 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"/>

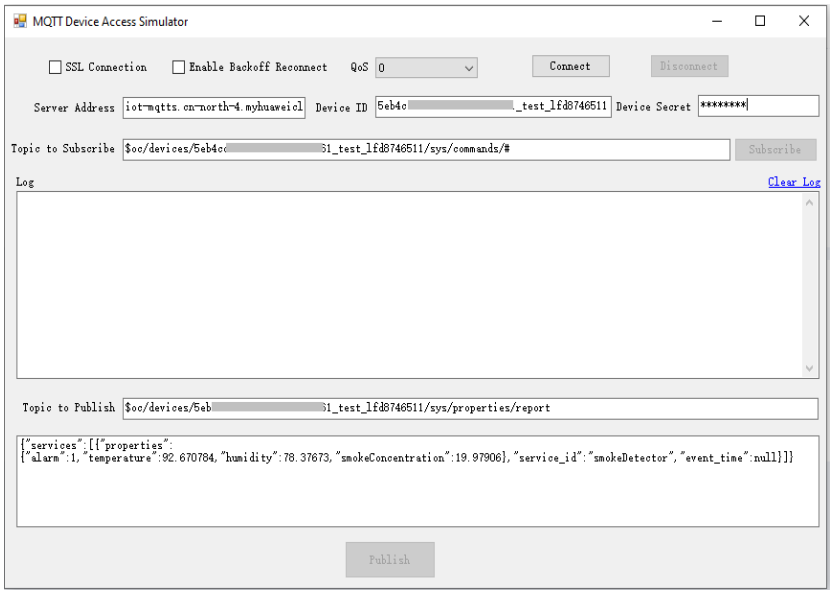
  <add key="deviceSecret" value="secret"/>

  <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);
        })));
    }
}
```

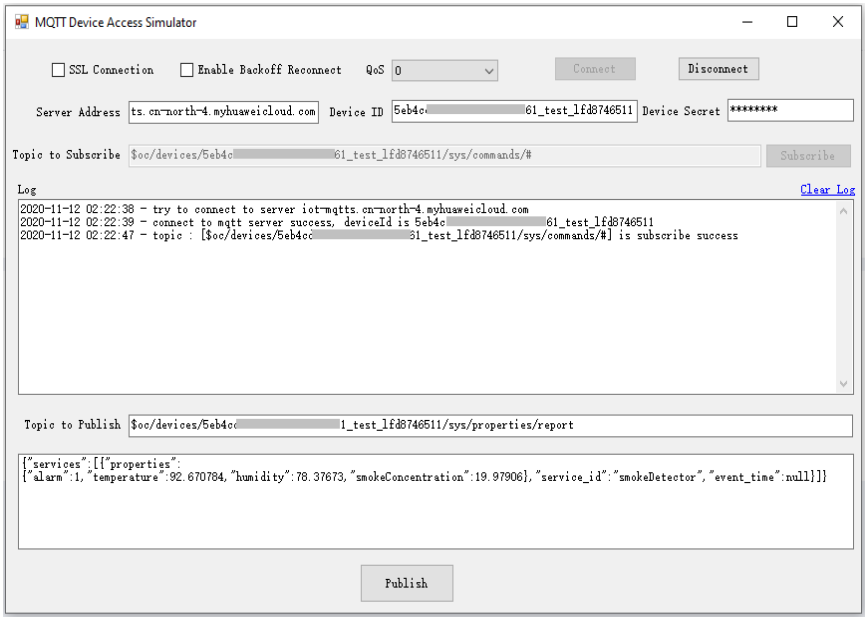
Subscribing to a Topic

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.

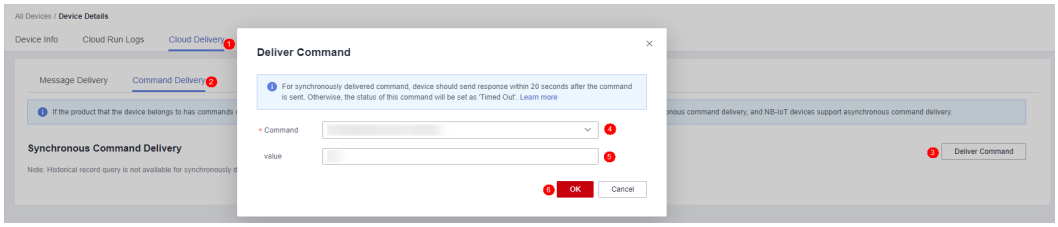
```
private void MqttApplicationMessageReceived(MqttApplicationMessageReceivedEventArgs e)
{
    Invoke((new Action(() =>
    {
```



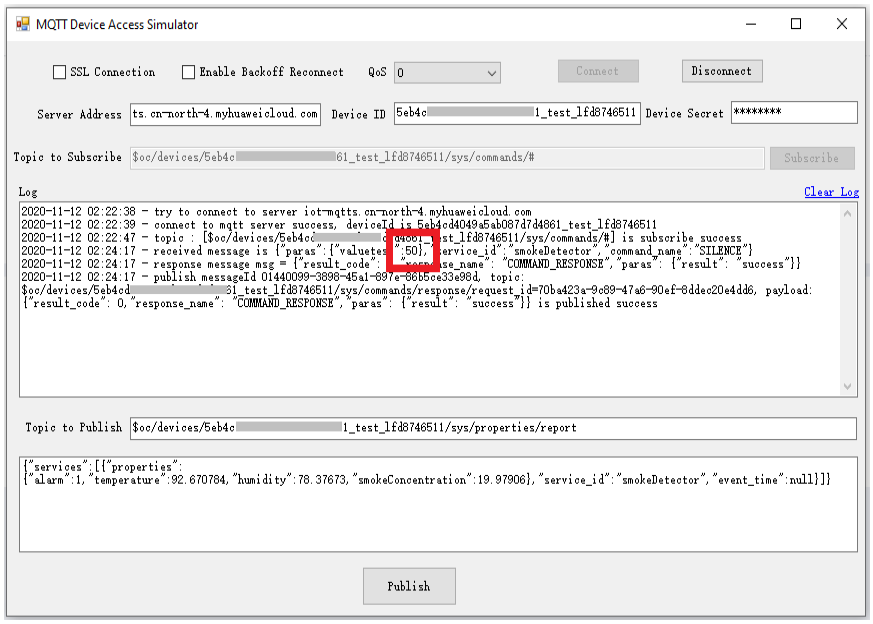
```
        ShowLogs($"received message is {Encoding.UTF8.GetString(e.ApplicationMessage.Payload)}
{Environment.NewLine}");

        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 4-17 Synchronous command delivery



After the command is delivered, the following information is displayed on the demo page:



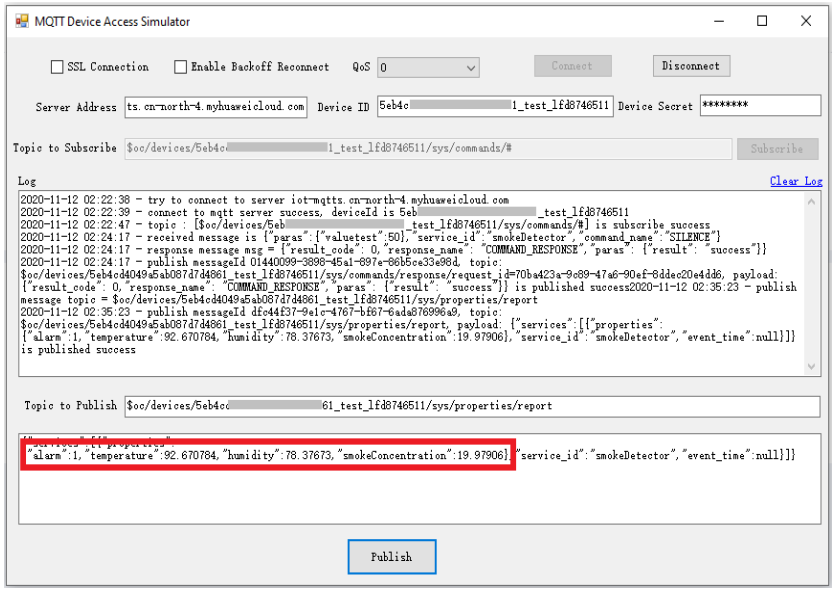
Publishing a Topic

Publishing a topic means that a device proactively reports its properties or messages to the platform. For details, see the API **Device Reporting Properties**.

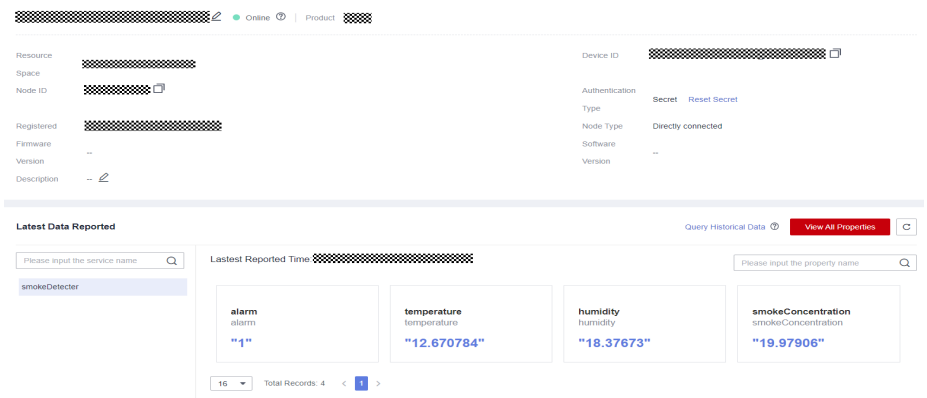
The **FrmMqttDemo** class implements the property reporting topic and property reporting.

```
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.



NOTE

If no latest data is displayed on the device details page, modify the services and properties in the product model to ensure that the reported services and properties are the same as those defined in the product model. Alternatively, go to the **Products** > **Model Definition** page and delete all services.

NOTE

Synchronous commands require device responses. For details, see **Upstream Response Parameters**.

4.3.7 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

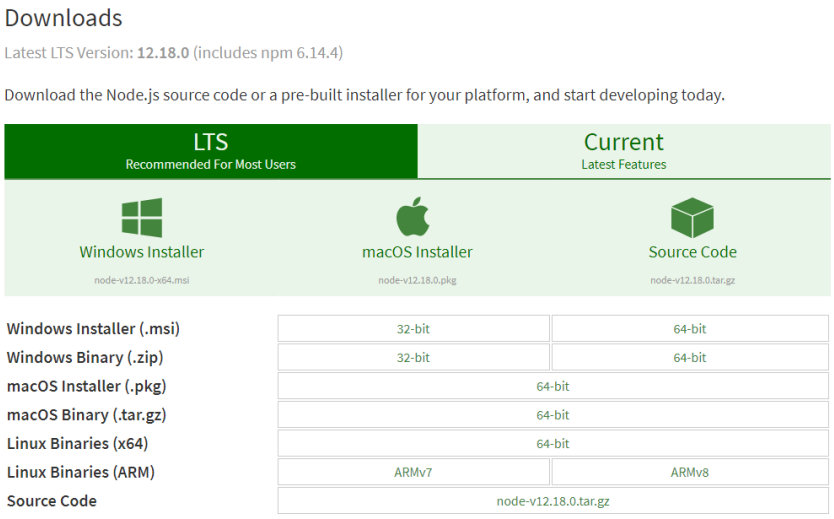
The code snippets in this document are only examples and are for trial use only. To put them into commercial use, obtain the IoT Device SDKs of the corresponding language for integration by referring to **Obtaining Resources**.

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.

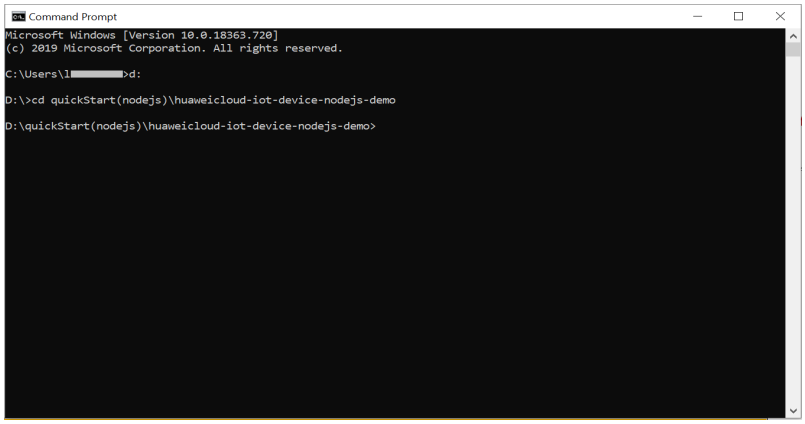
Step 2 Press **Win+R**, enter **cmd**, and press **Enter** to open the CLI. Run the following commands to install the global module:

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
var serverUrl = "****"; // Enter the access address of the platform that the device is connected to.
var deviceId = "****";// Enter the ID of the device registered with the platform.
var secret = "****";// Enter the secret of the device registered with the platform.
var timestamp = dateFormat("YYYYmmddHH", new Date());

var propertiesReportJson = {'services':[{'properties':
{'alarm':1,'temperature':12.670784,'humidity':18.37673,'smokeConcentration':19.97906},'service_id':'smokeDet
ector','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

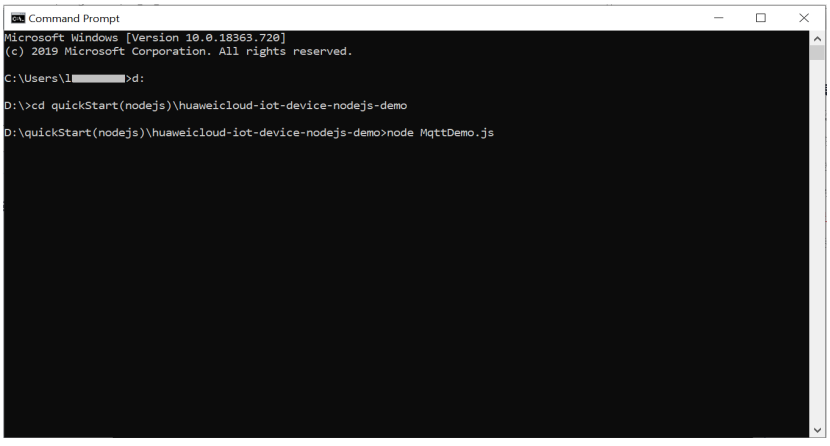
Starting the Demo

To connect a device or gateway to the platform, upload the device information to bind the device or gateway to the platform.

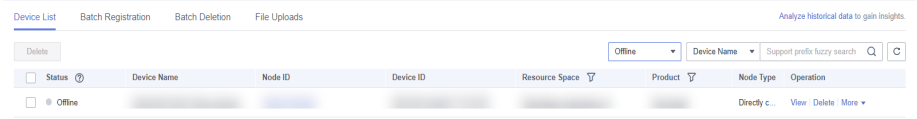
1. This demo provides methods such as establishing an MQTT or MQTTS connection. By default, MQTT uses port 1883, and MQTTS uses port 8883. (In the case of MQTTS connections, you must load the certificate for verifying the platform identity. The certificate is used for login authentication when the device 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('responsed message is ' + jsonMsg);
})
```

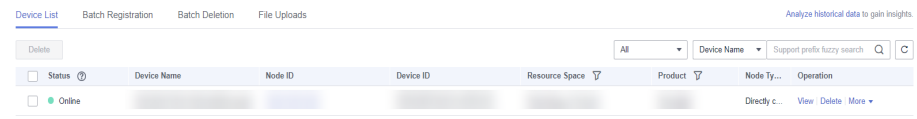
Find the Node.js demo source code directory, modify **key project parameters**, and start the demo.



Before the demo is started, the device is in the offline state.



After the demo is started, the device status changes to online.



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");
        }
    })
}
```

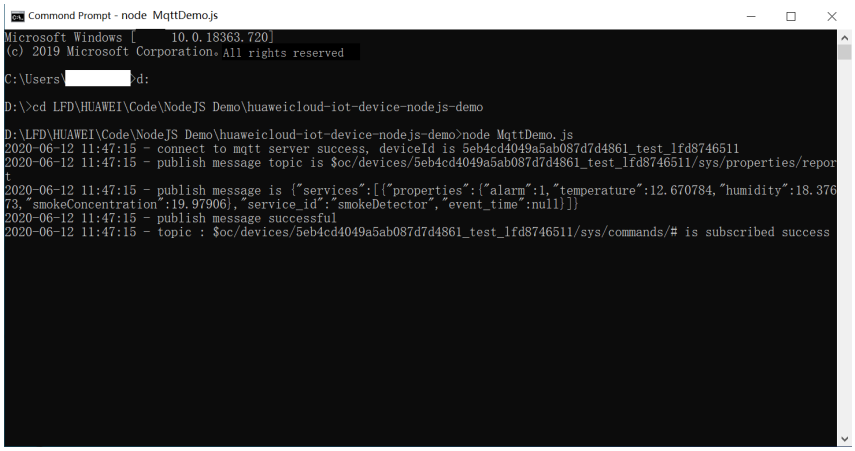
3. Publishing a topic means that a device proactively reports its properties or messages to the platform. For details, see the API **Device Reporting Properties**. After the connection is successful, call the **publishMessage** method to report properties.

```
// 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");
}
```

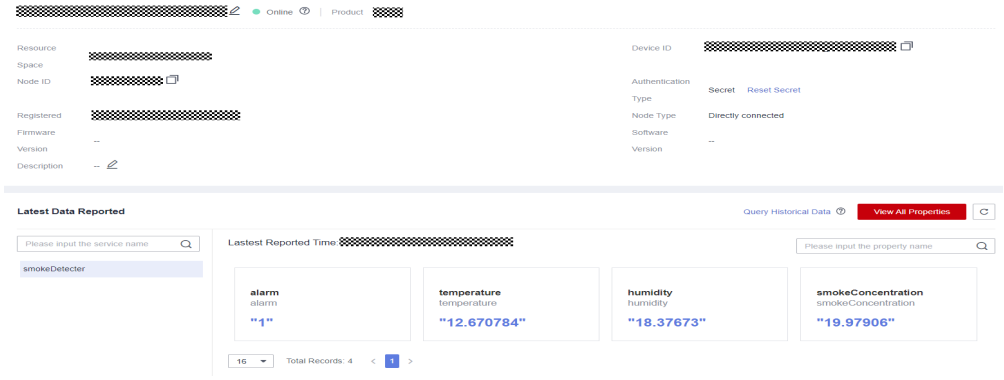
Reported properties in the JSON format are as follows:

```
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:



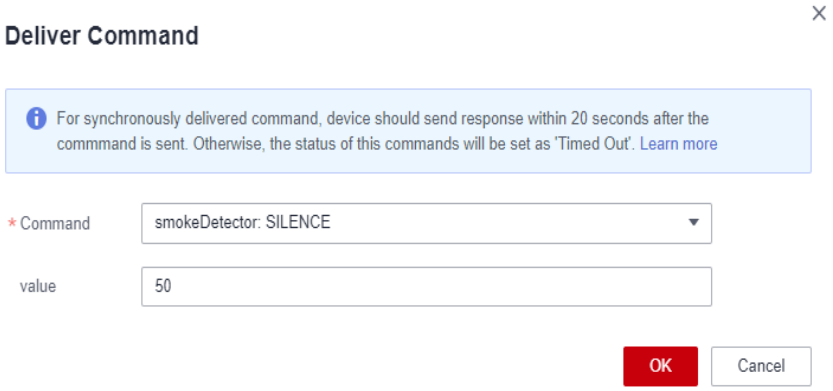
NOTE

If no latest data is displayed on the device details page, modify the services and properties in the product model to ensure that the reported services and properties are the same as those defined in the product model. Alternatively, go to the **Products** > **Model Definition** page and delete all services.

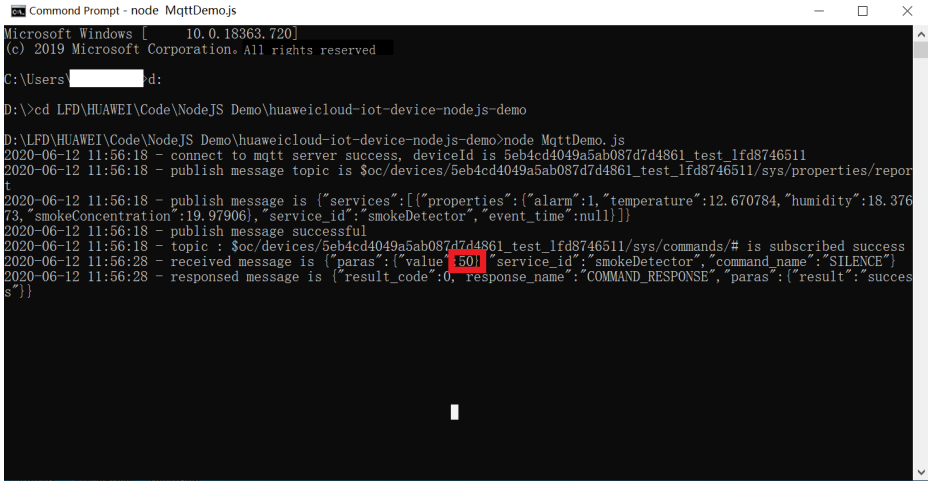
Receiving a Command

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 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**.



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**.

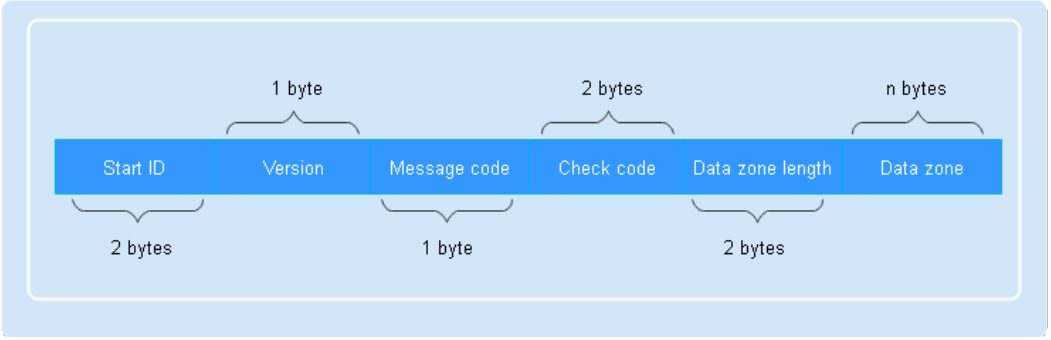
4.4 OTA Upgrade Adaptation on the Device Side

4.4.1 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.

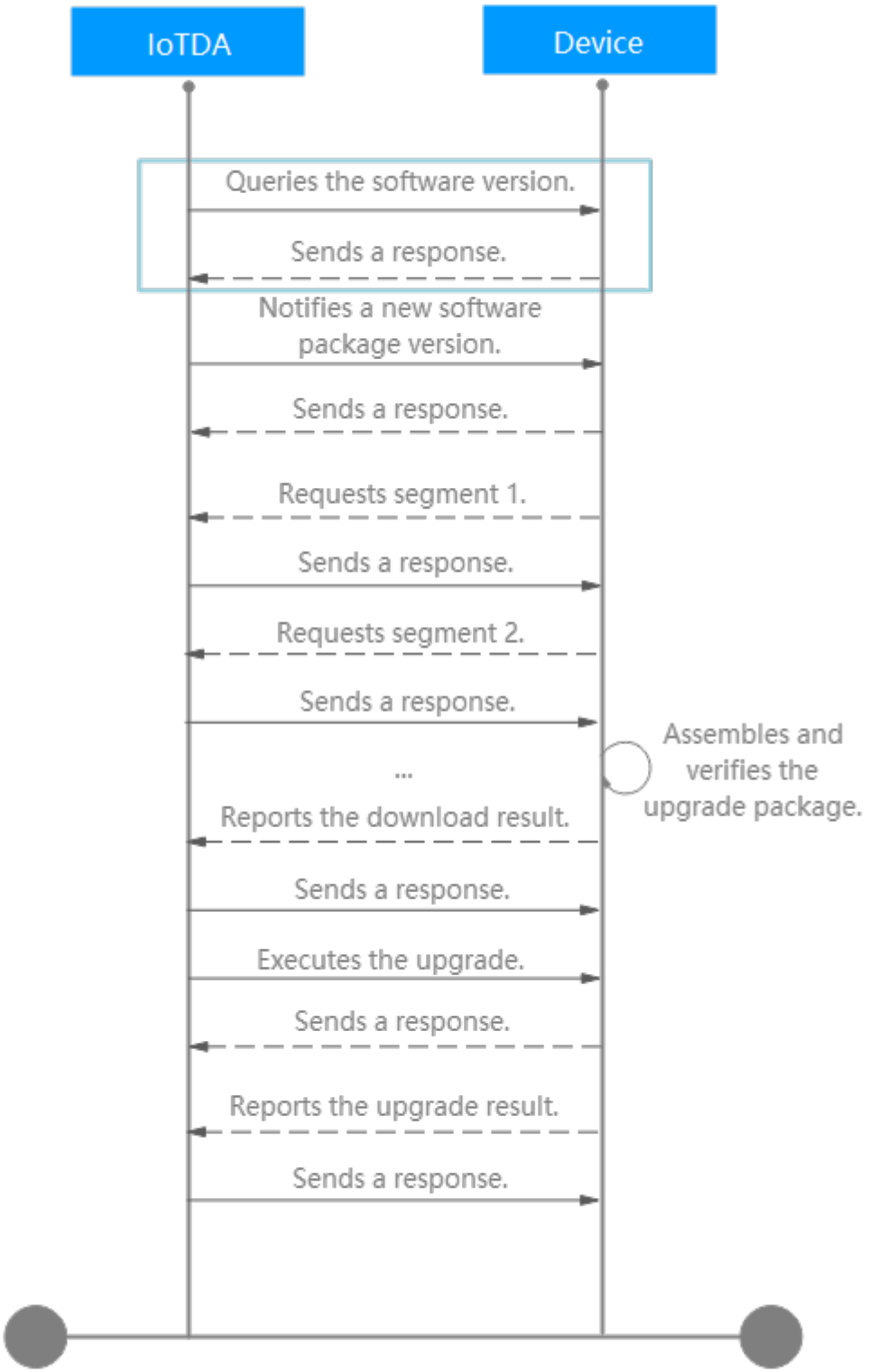
Field	Type	Description
Start ID	WORD	The value is fixed at **0XFFFE**.
Version	BYTE	The four most significant bits are reserved. The four least significant bits indicate the protocol version. Currently, the version is **1**.
Message code	BYTE	Type of the request exchanged between the platform and device. 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.

Field	Type	Description
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
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.

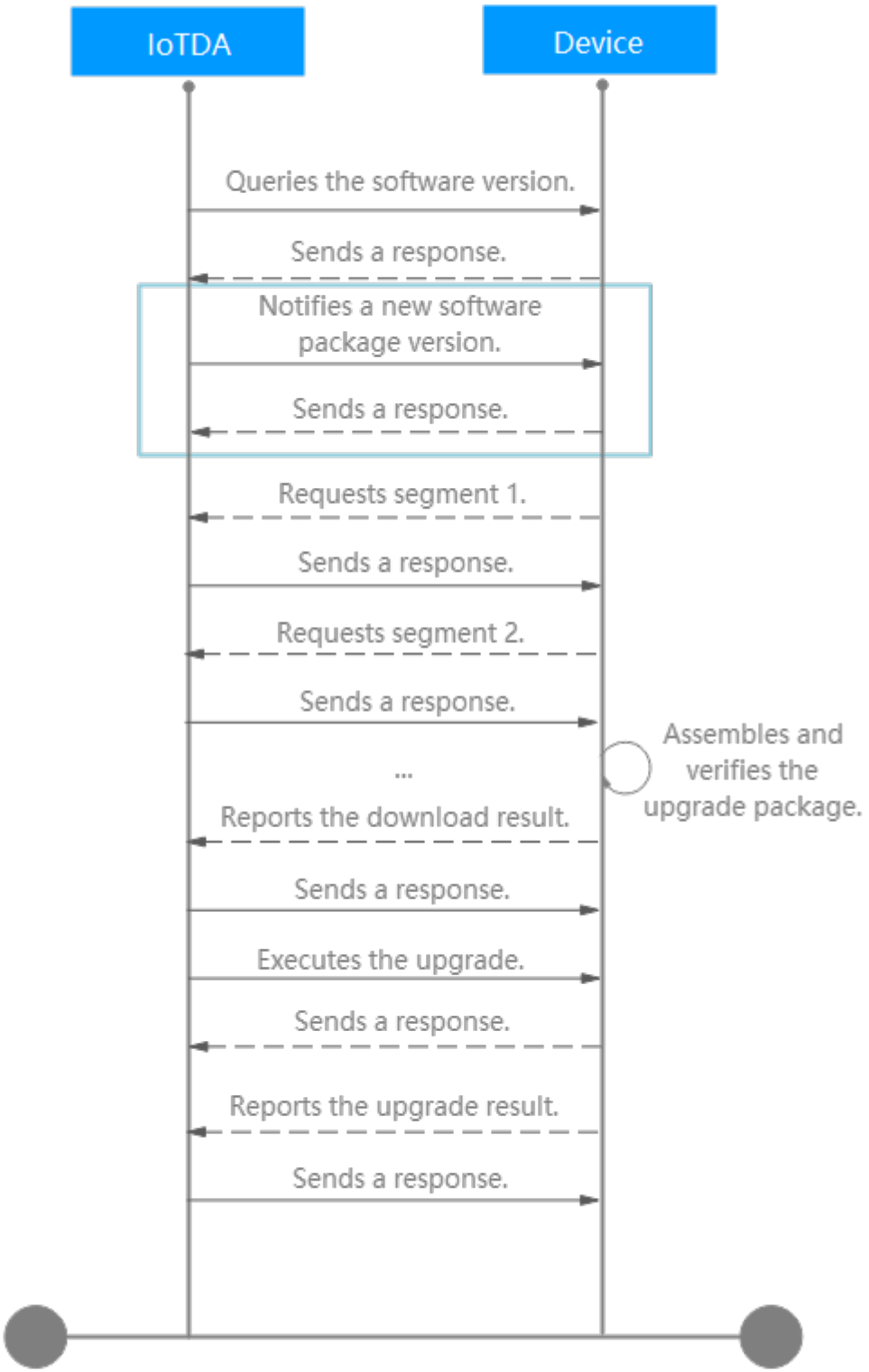
Field	Data Type	Description
Result code	BYTE	The value is **0X00**, indicating that the processing was successful.

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.

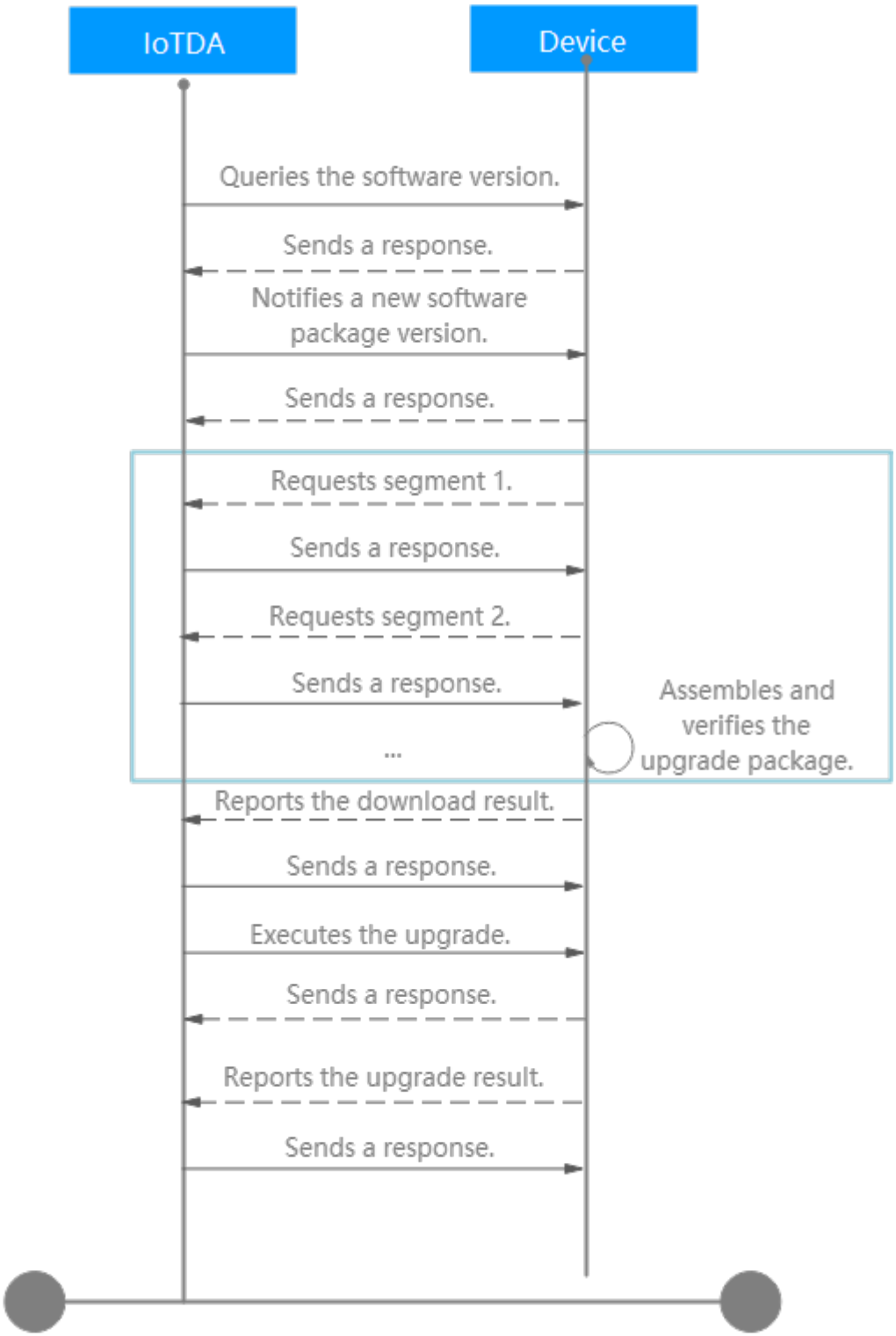
- **Message code**: The value is 14 (the same as that in the request).
- **Check code**: The value 0000 is used before CRC16 calculation.
- **Data zone length**: In accordance with the data type of the fields in the data zone, the length is 1 byte (hexadecimal value: 0001).
- **Data zone**: The device 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. 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 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.

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.

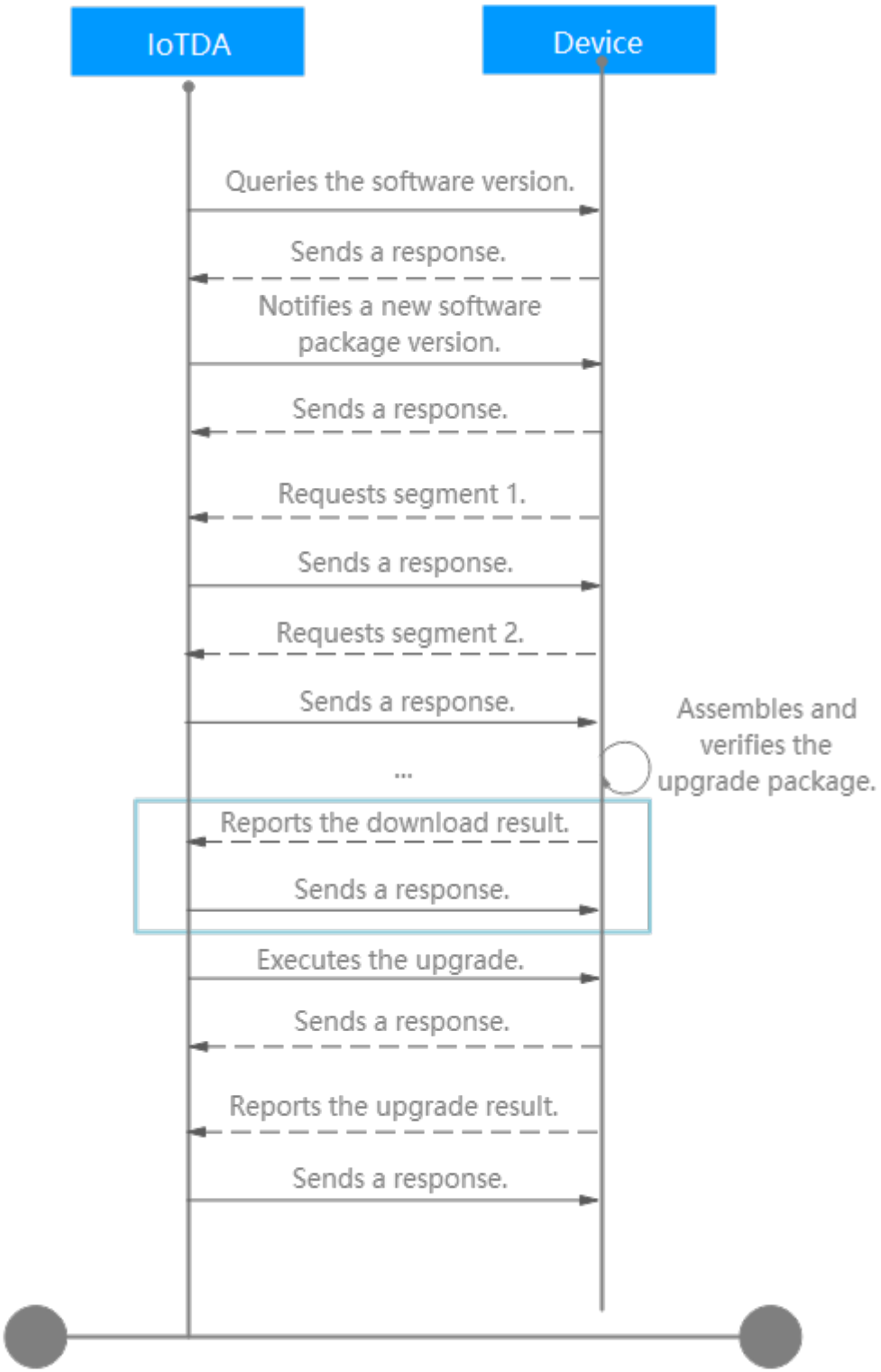
- **Data zone length**: In accordance with the data type of the fields in the data zone, the length is 19 bytes (hexadecimal value: 0013).

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:

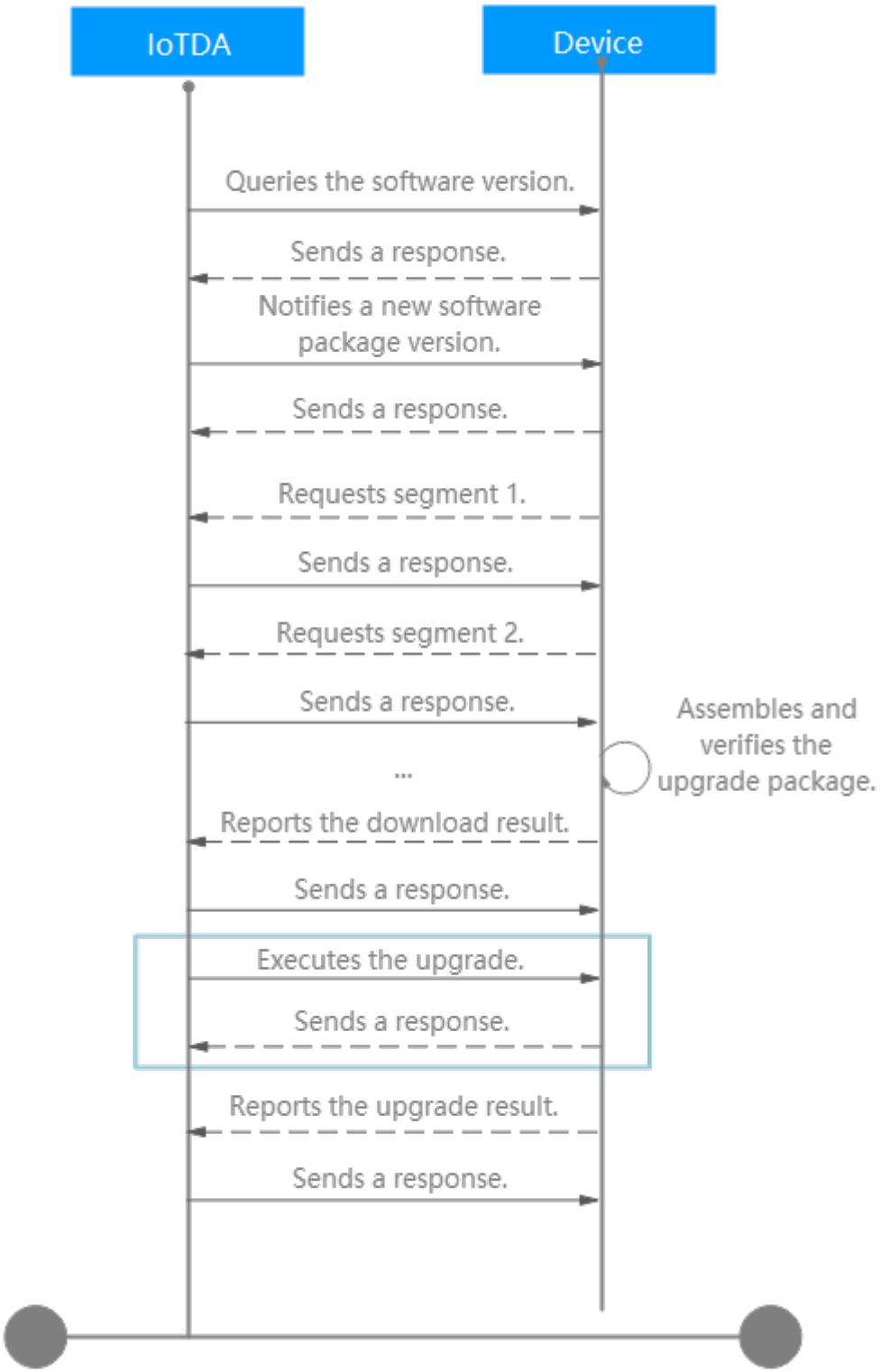
- **Start ID**: The value is fixed at FFFE.
- **Version**: The value is fixed at 01.
- **Message code**: The value is 16 (the same as that in the request).
- **Check code**: The value 0000 is used before CRC16 calculation.
- **Data zone length**: In accordance with the data type of the fields in the 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:

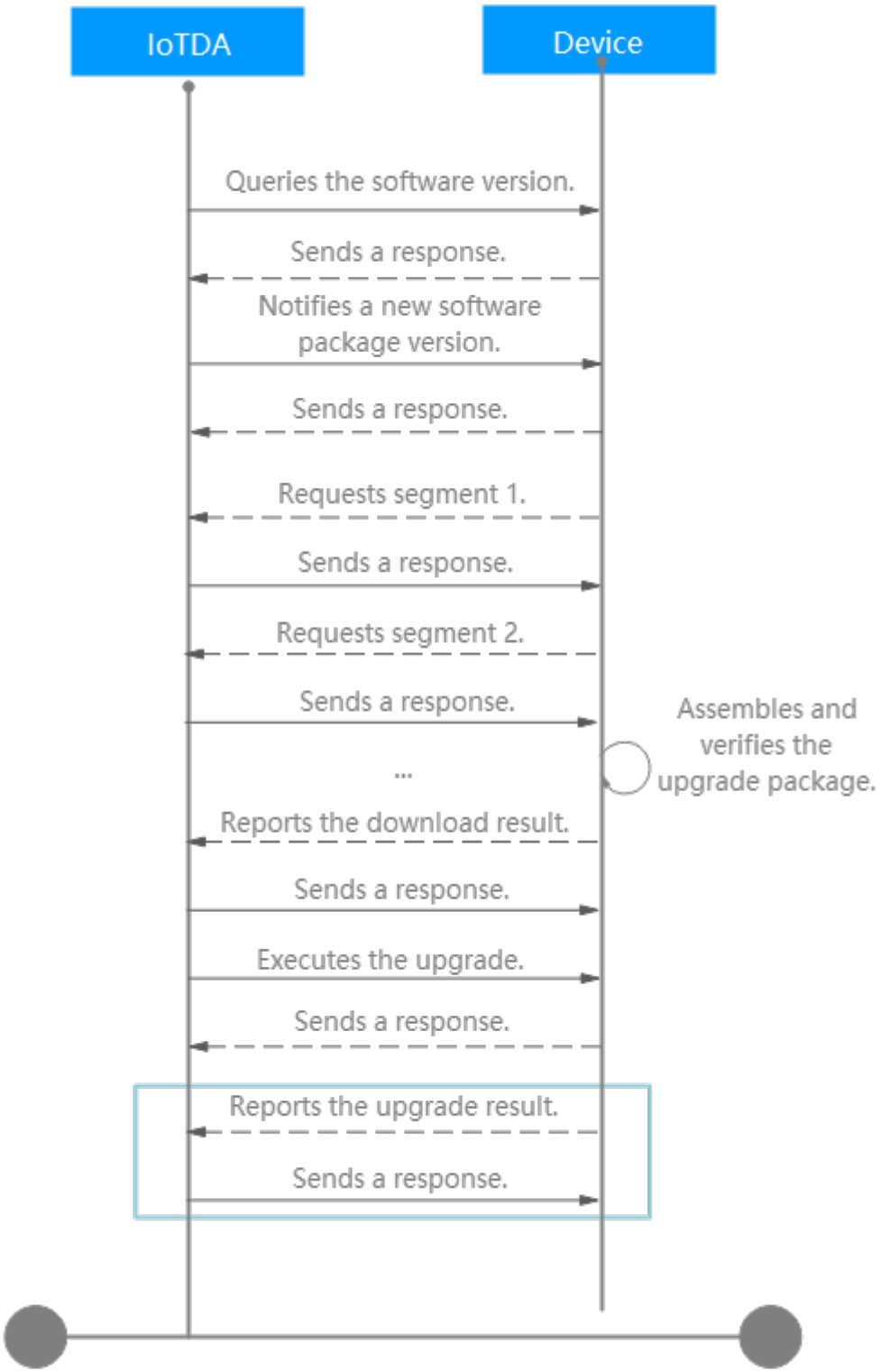
- **Start ID**: The value is fixed at FFFE.
- **Version**: The value is fixed at 01.
- **Message code**: The value is 17 (the same as that in the request).
- **Check code**: The value 0000 is used before CRC16 calculation.
- **Data zone length**: In accordance with the data type of the fields in the 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.

Now, the adaptation of the software upgrade is complete.

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);
}
```

4.4.2 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	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**.
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

Field	Type	Description
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	The value is **0X00**, indicating that 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.
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.

Field	Data Type	Description
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.

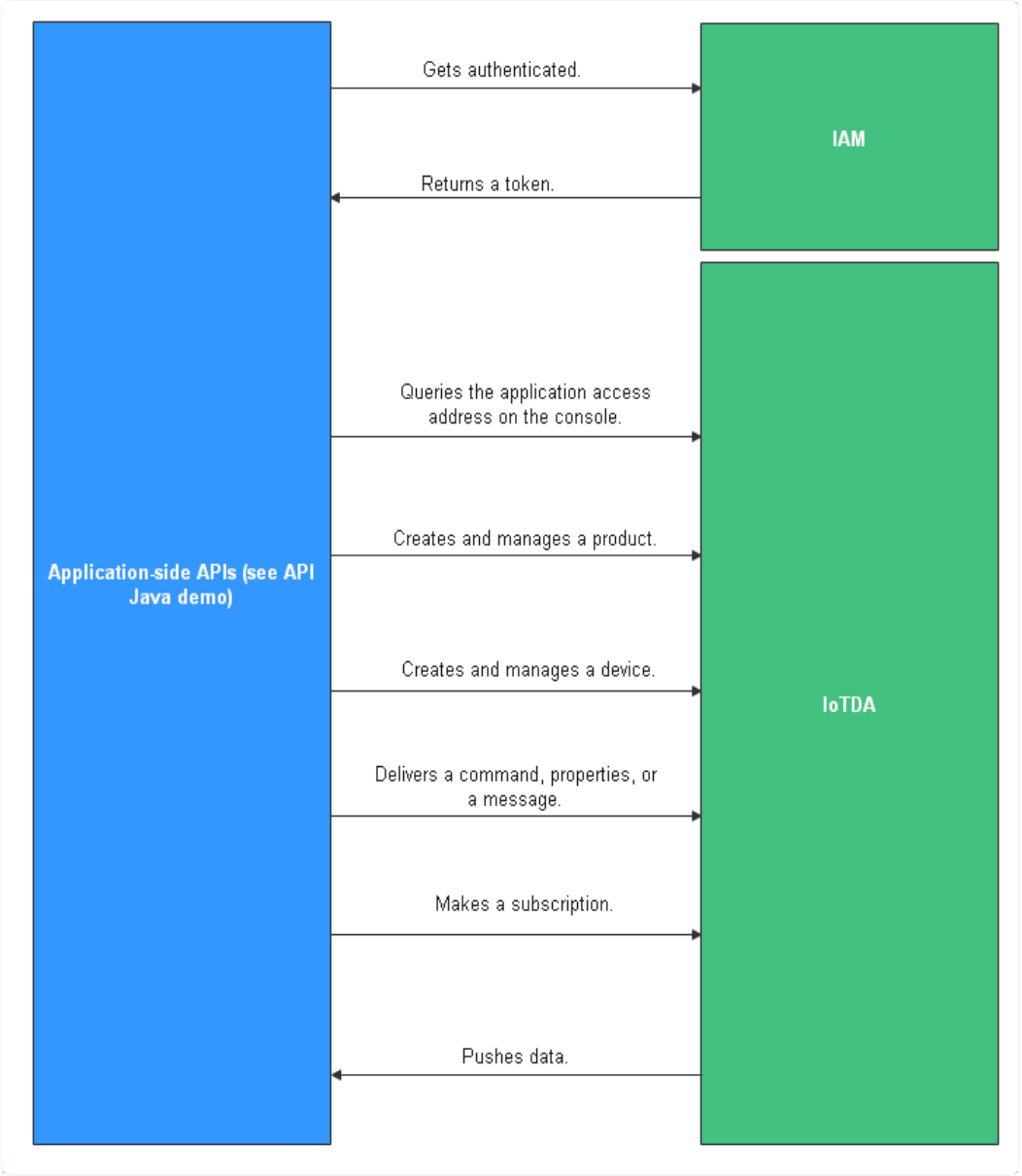
5 Development on the Application Side

5.1 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.

NOTICE

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**.	**Java SDK**
Application C# SDK	You can use C# methods to call **application-side APIs** to communicate with the platform. For details, see **C# SDK**.	**C# SDK**
Application Python SDK	You can use Python methods to call **application-side APIs** to communicate with the platform. For details, see **Python SDK**.	**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**.	**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**.	**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**.	**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 stacking policy management	Used by applications to manage stacking policies, including creating, querying, modifying, and deleting stacking policies.
Data flow control policy management	Used by applications to manage flow control policies, including creating, querying, modifying, and deleting flow control policies.

5.2 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**

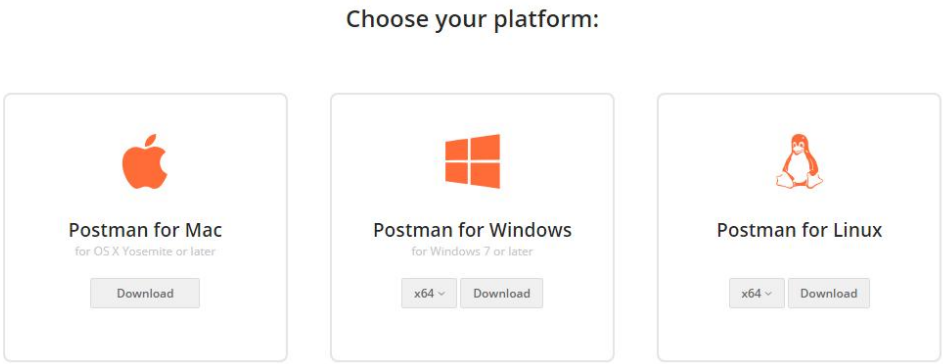
Prerequisites

- 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.



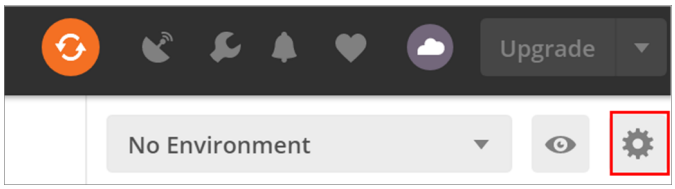
NOTE

- Postman requires the **.NET Framework 4.5** component. **Download it**.
- To ensure successful API calls, you are advised to download **the latest version of Postman (32-bit) for Windows**.

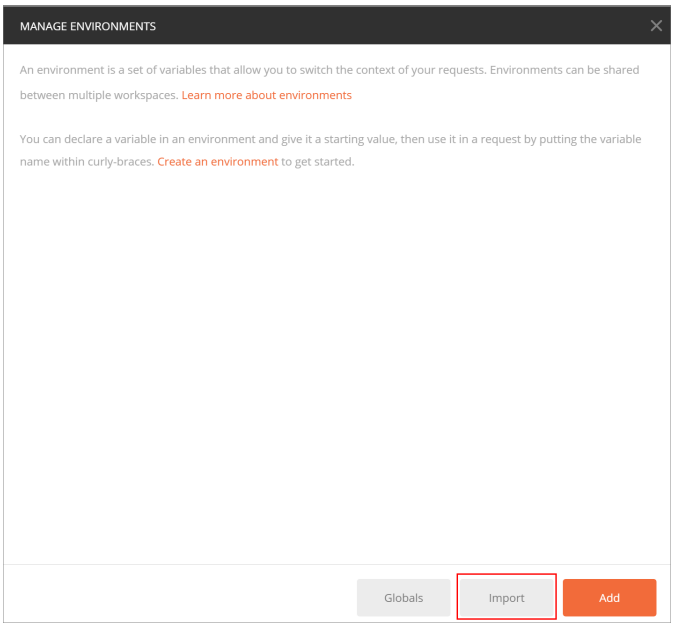
2. Enter the email address, username, and password to register Postman.

Step 2 Import the Postman environment variables.

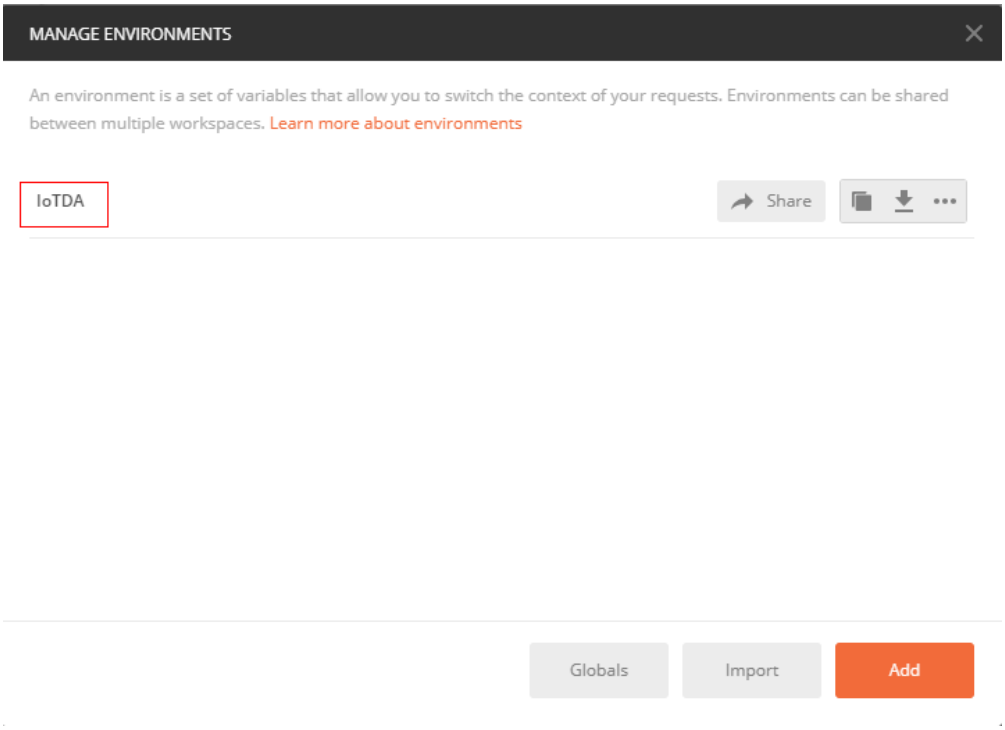
1. Click  in the upper right corner to open the **MANAGE ENVIRONMENTS** window.



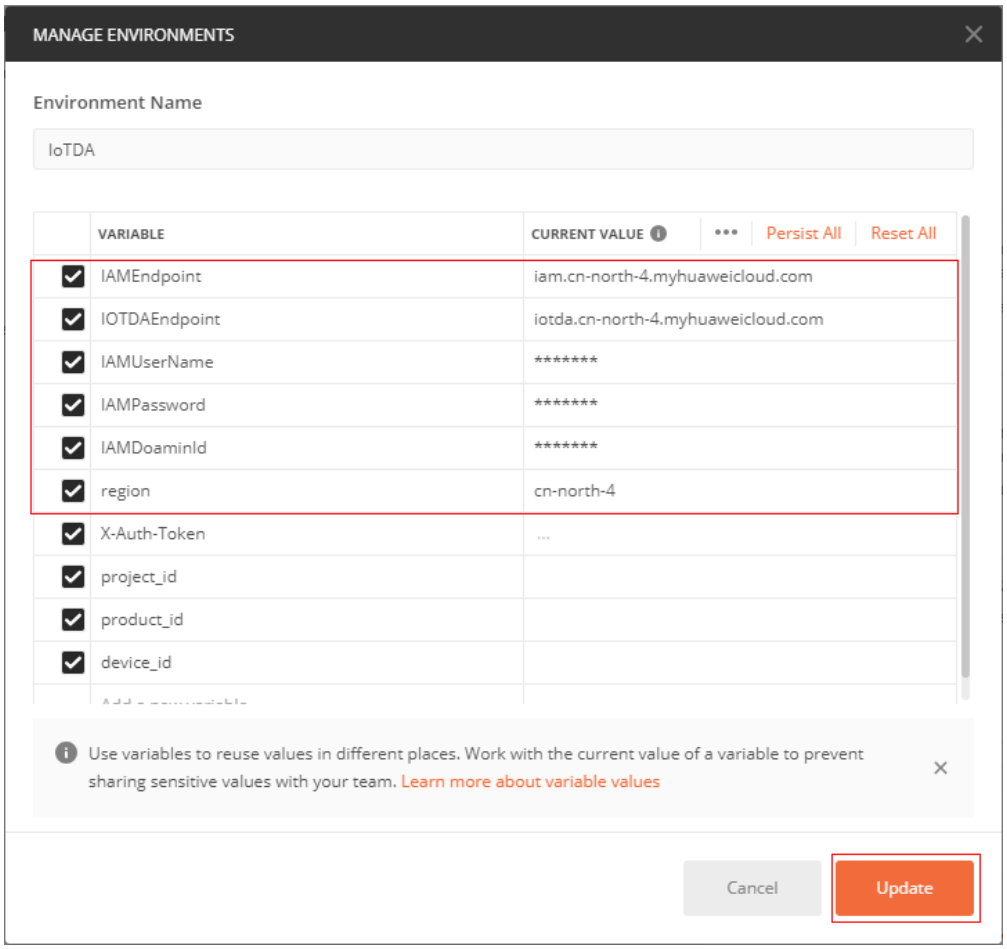
2. Click **Import**. On the page displayed, click **Select File** to import the **IoTDA.postman_environment.json** file (obtained after the **Collection** package is decompressed).



3. Click the **IoTDA** environment imported.



4. Configure parameters based on the following table.

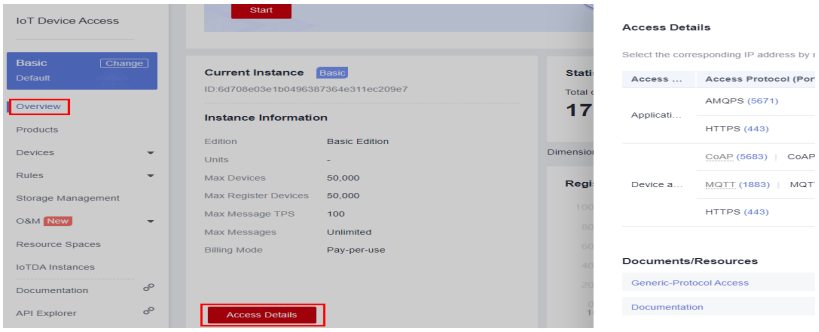


Parameter	Description
IAMEndpoint	IAM endpoint. For details, see **Regions and Endpoints**.
IoTDAEndpoint	IoTDA endpoint. For details, see **Step 2.5**.
IAMUserName	IAM username, which can be obtained from the **My Credentials** page.
IAMPassword	Password for logging in to Huawei Cloud.
IAMDoaminId	Account name, which can be obtained from the **My Credentials** page.
region	Region where IoTDA is enabled.

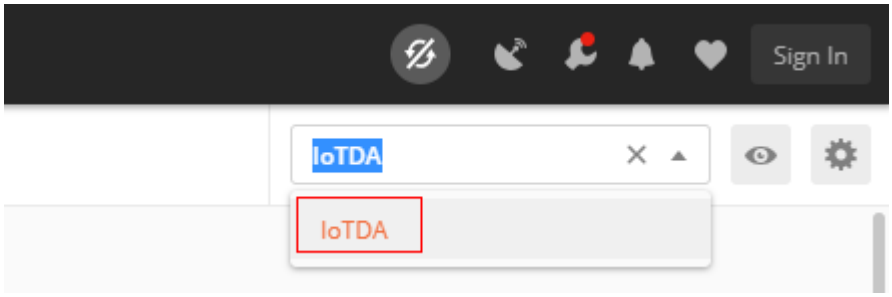
5. Obtain IoTDA endpoints.

   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.

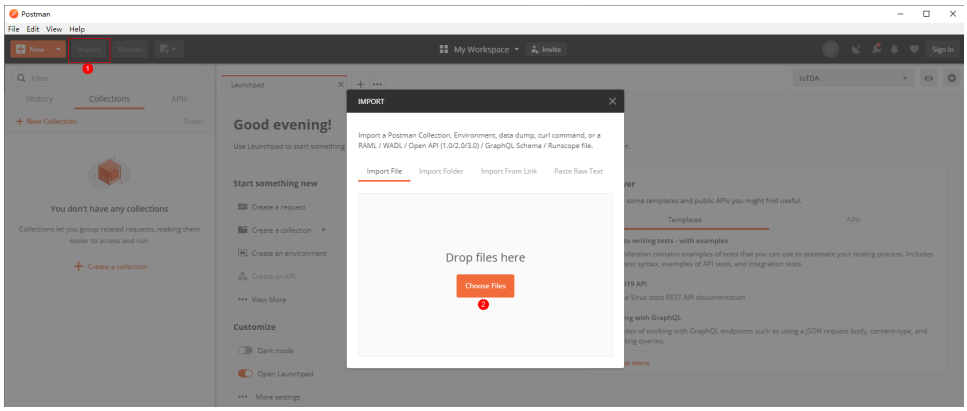
Figure 5-1 Access details



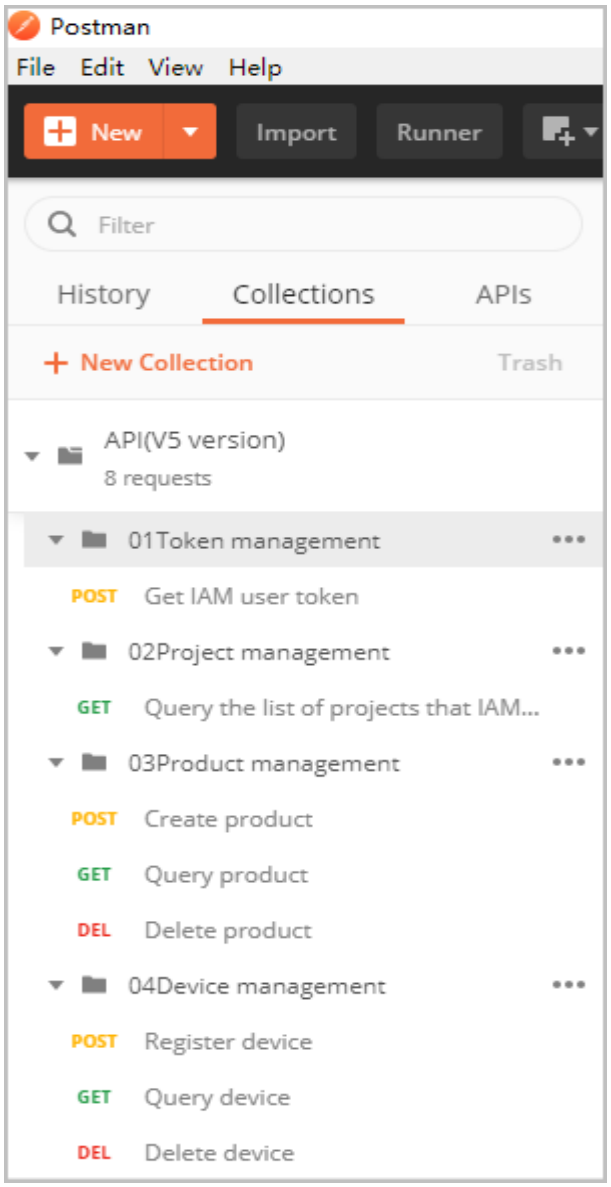
6. Return to the home page and set the environment variable to the imported IoTDA.



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:

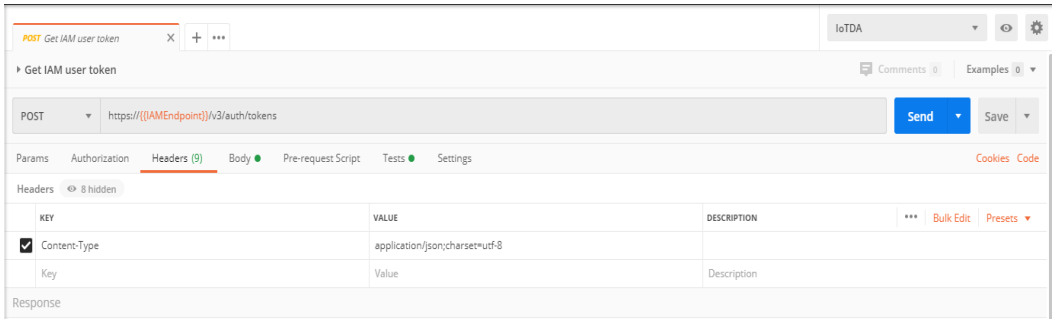
```
POST https://iam.cn-north-4.myhuaweicloud.com/v3/auth/tokens
Content-Type: application/json

{
    "auth": {
        "identity": {
            "methods": [
                "password"
            ],
```



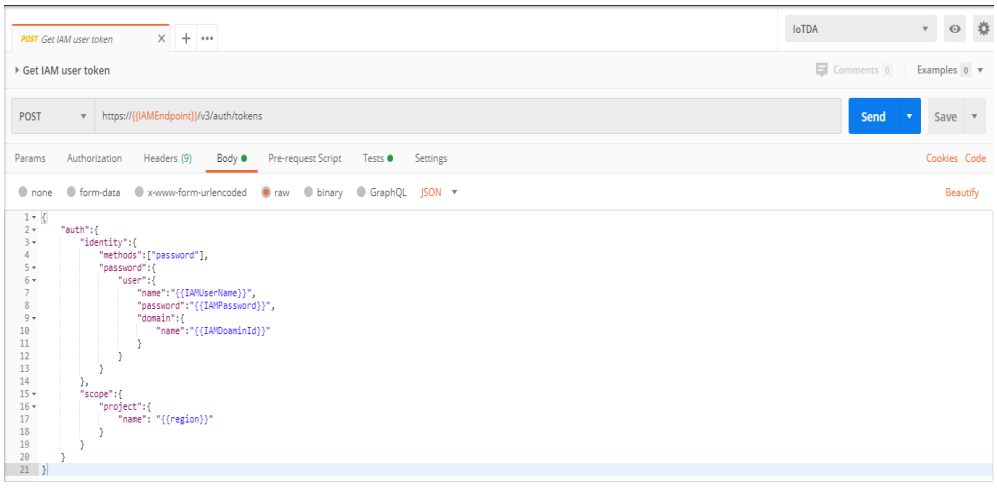
```
            "password": {
                "user": {
                    "name": "username",
                    "password": "********",
                    "domain": {
                        "name": "domainname"
                    }
                }
            }
        },
        "scope": {
            "project": {
                "name": "xxxxxxxx"
            }
        }
    }
}
```

Debug the API by following the instructions provided in **Obtaining the Token of an IAM User**.

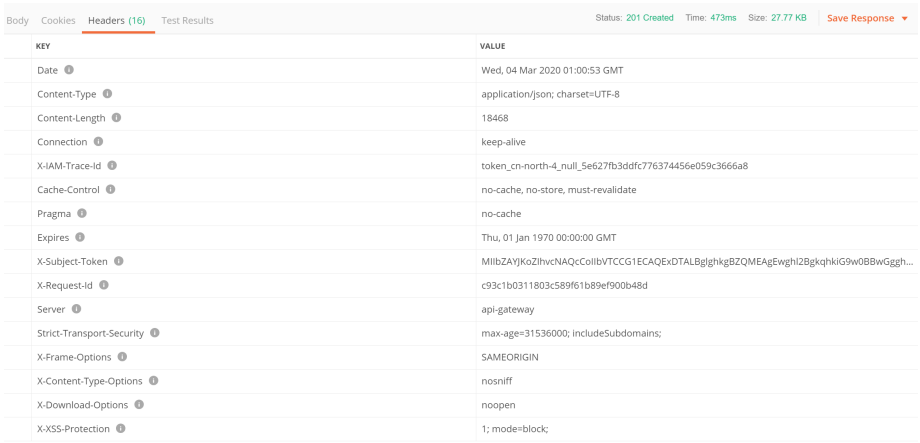
Step 1 Configure the HTTP method, URL, and headers of the API.



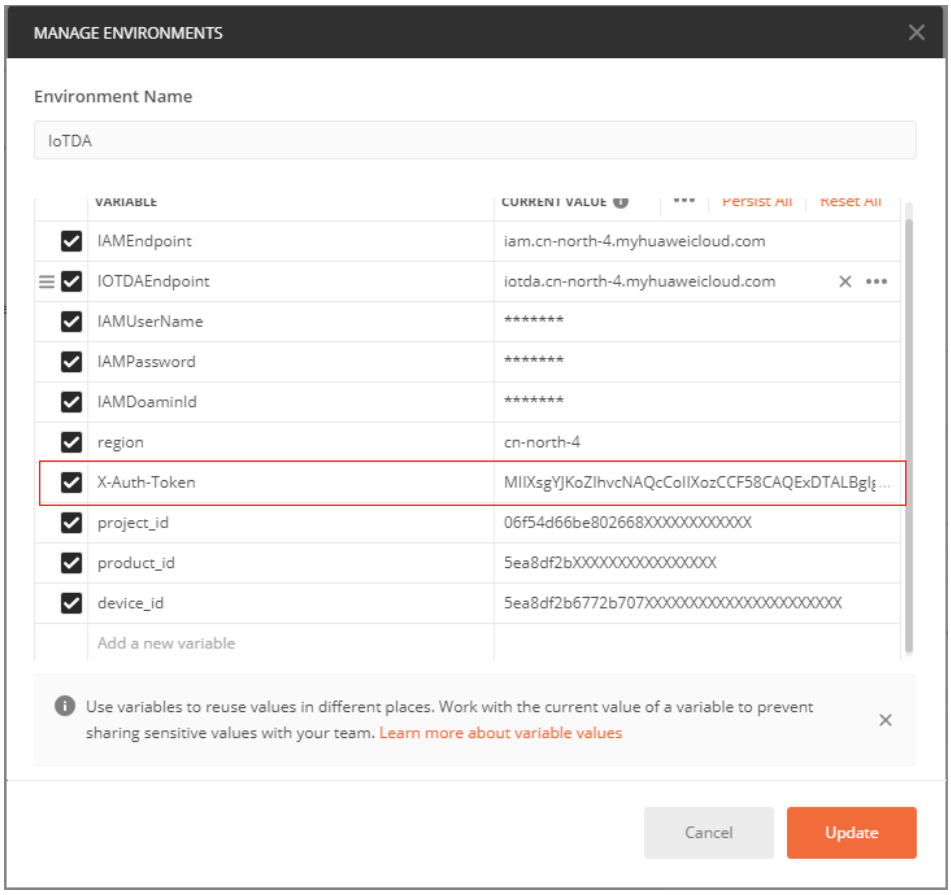
Step 2 Configure the body of the API.



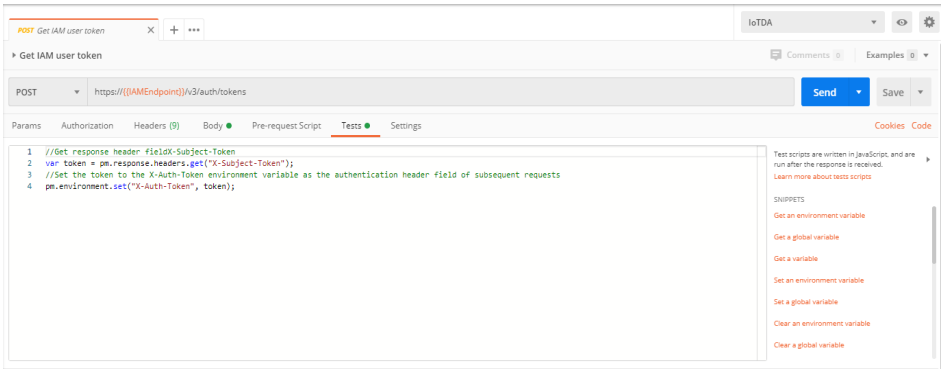
Step 3 Click **Send**. The returned code and response are displayed in the lower part of the page.



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

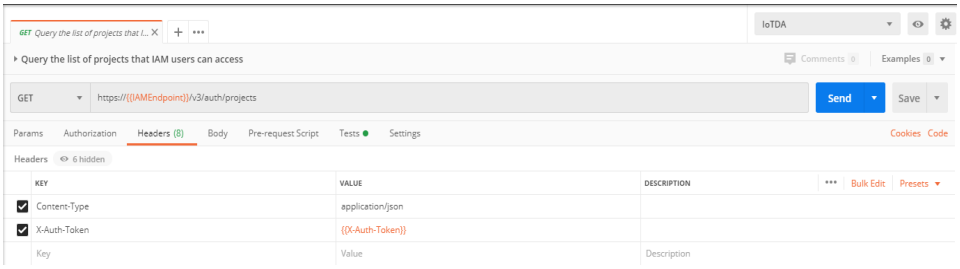
Before accessing platform APIs, the application must call the API **Listing Projects Accessible to an IAM User** to obtain the project ID of the user.

To call this API, the application constructs an HTTP request. An example request is as follows:

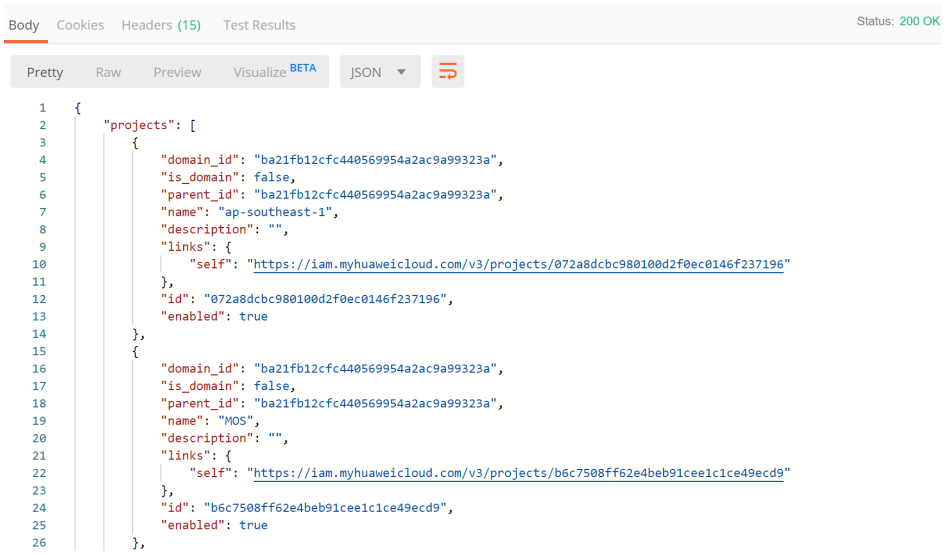
```
GET https://iam.cn-north-4.myhuaweicloud.com/v3/auth/projects
Content-Type: application/json
X-Auth-Token: ********
```

Debug the API by following the instructions provided in **Listing Projects Accessible to an IAM User**.

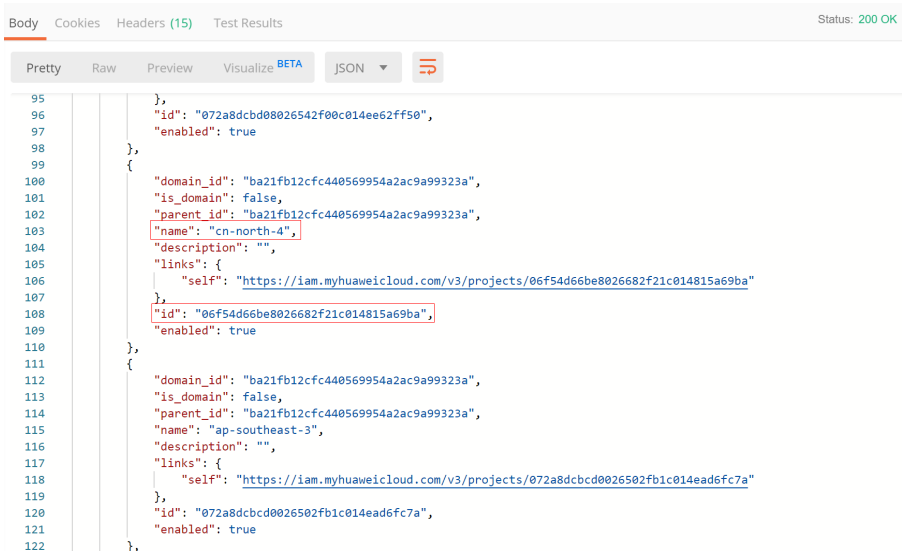
Step 1 Configure the HTTP method, URL, and headers of the API.

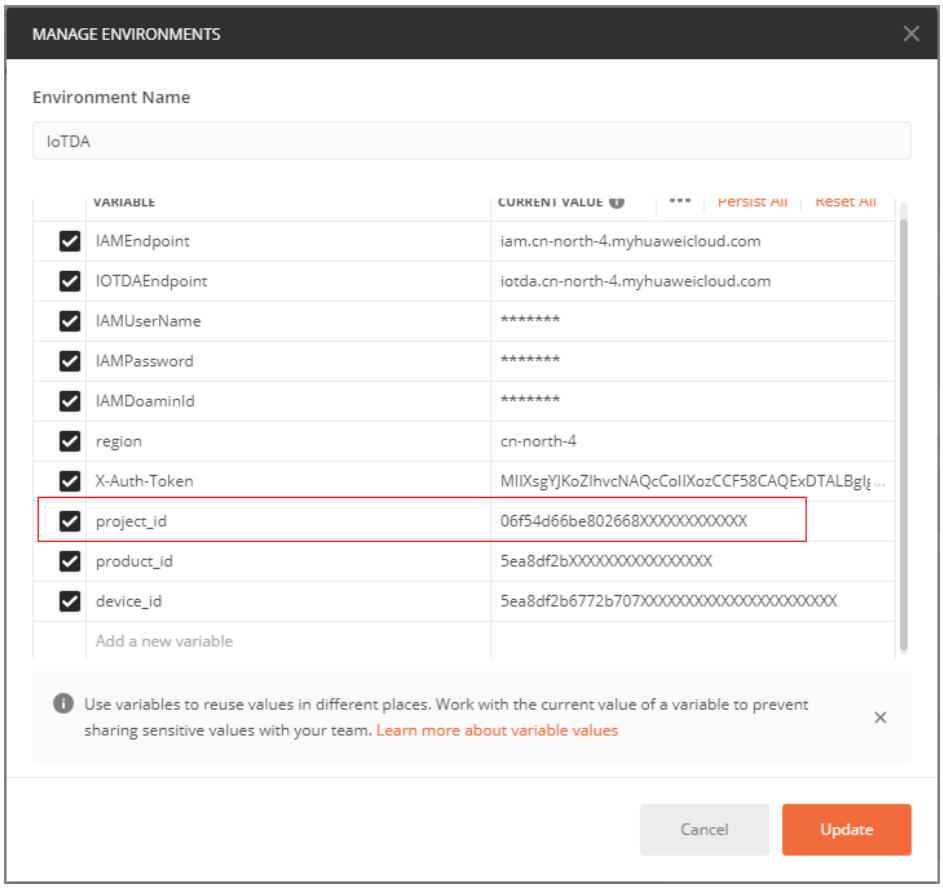


Step 2 Click **Send**. The returned code and response are displayed in the lower part of the page.

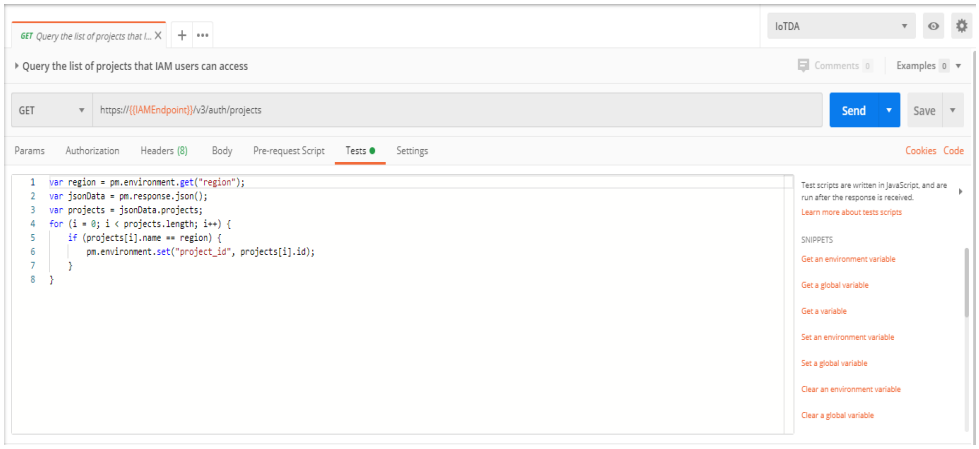


Step 3 The returned body contains a list of projects. Search for the item whose **name** is the same as the value of **region** in the IoTDA environment, and use the **id** value to update **project_id** in the IoTDA environment so that it can be used in other API calls.





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.

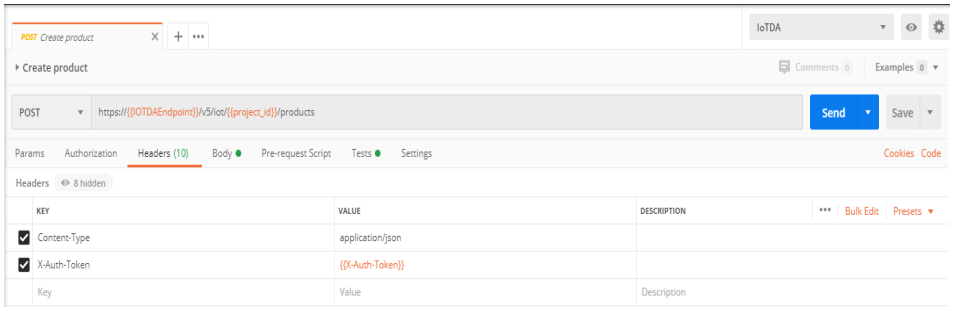
To call this API, the application constructs an HTTP request. An example request is as follows:


```
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"
}
```

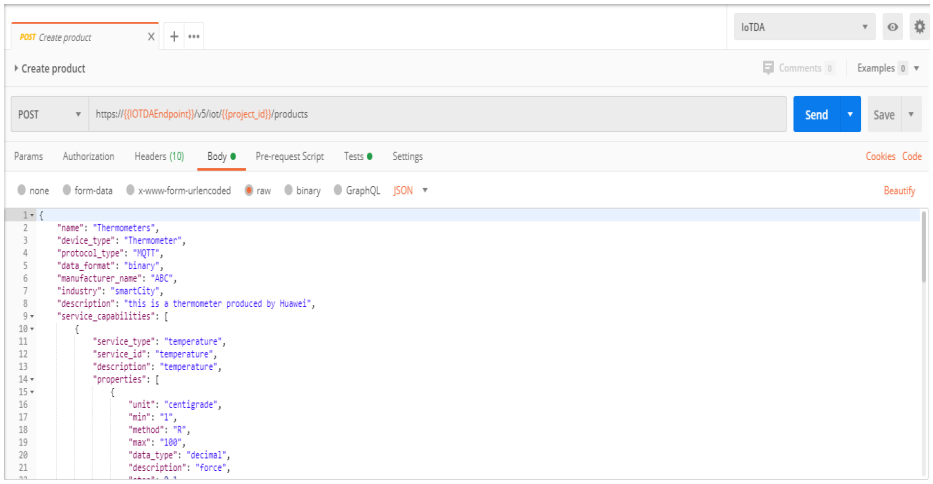
Debug the API by following the instructions provided in **Creating a Product**.

Note: Only the parameters used in the debugging example are described in the following steps.

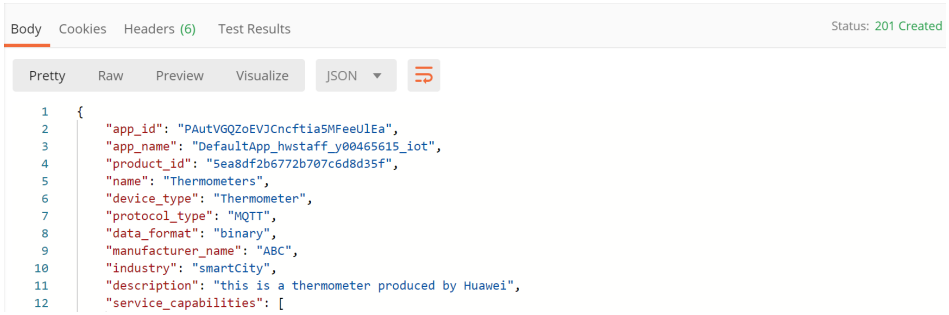
Step 1 Configure the HTTP method, URL, and headers of the API.



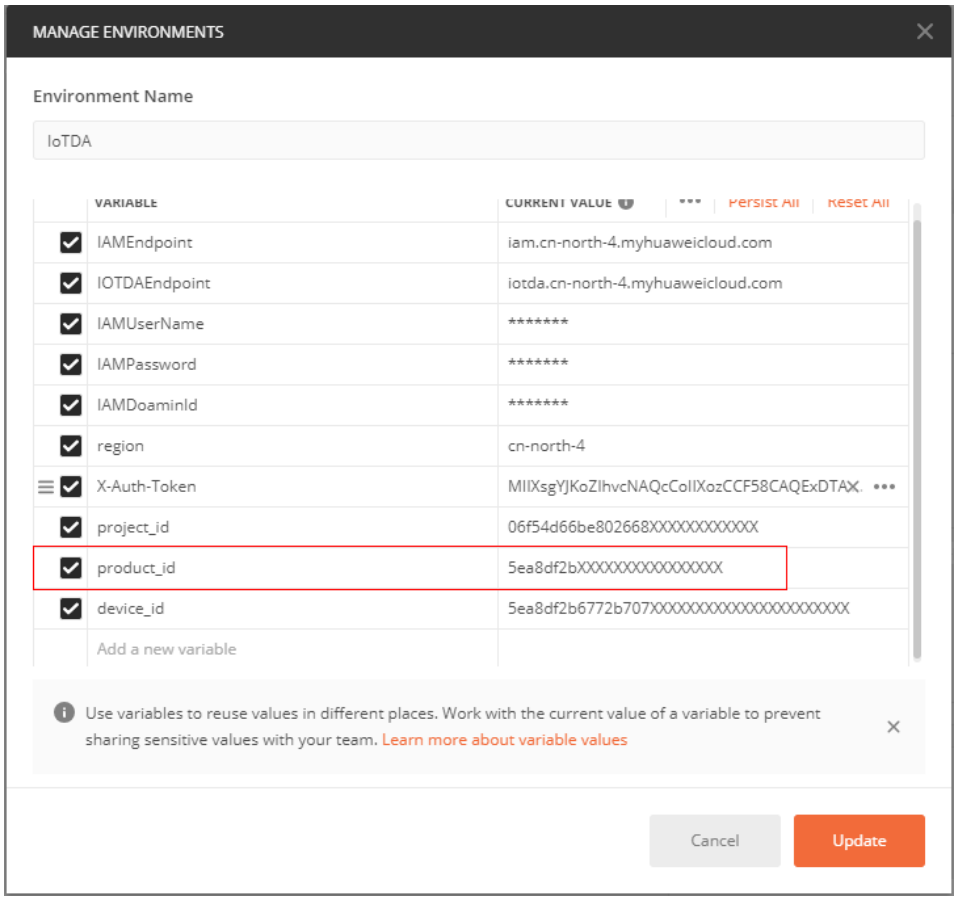
Step 2 Configure the body of the API.



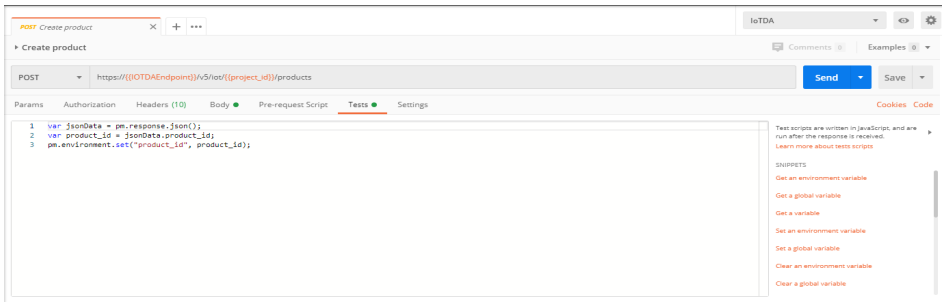
Step 3 Click **Send**. The returned code and response are displayed in the lower part of the page.



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.

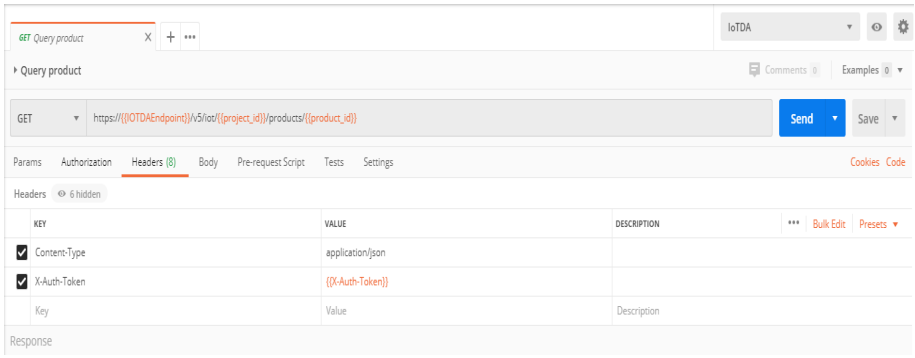
To call this API, the application constructs an HTTP request. An example request is as follows:

```
GET https://iotda.cn-north-4.myhuaweicloud.com/v5/iot/{project_id}/products/{product_id}
Content-Type: application/json
X-Auth-Token: ********
```

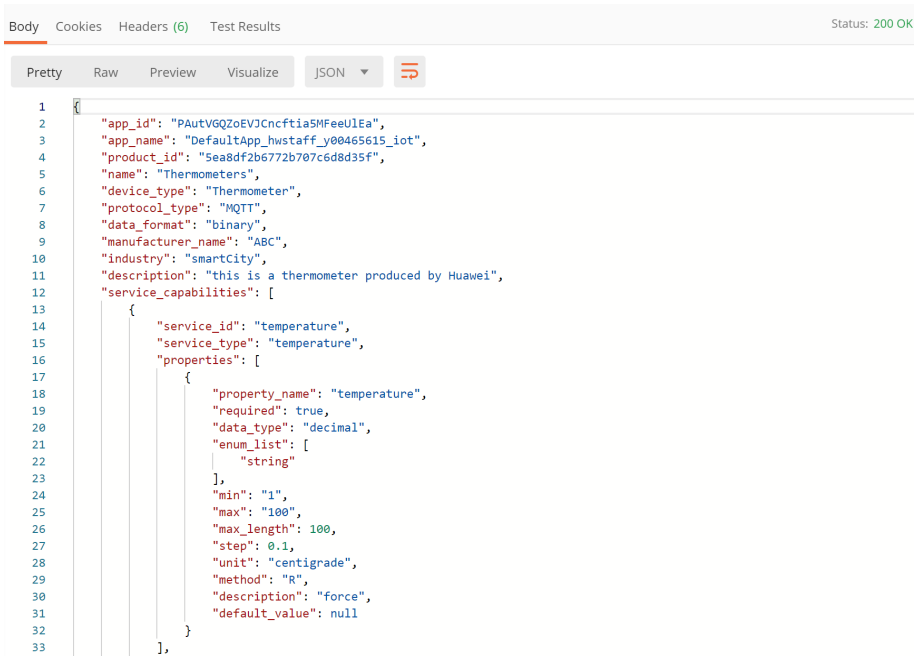
Debug the API by following the instructions provided in **Querying a Product**.

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",
```

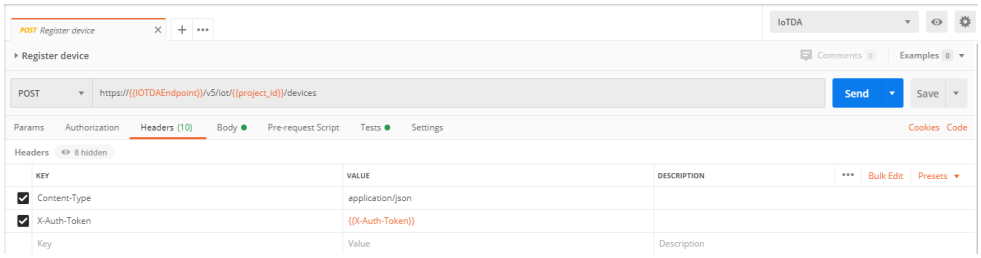


```
 "device_name" : "dianadevice",
 "product_id" : "b640f4c203b7910fc3cbd446ed437cbd",
 "auth_info" : {
   "auth_type" : "SECRET",
   "secure_access" : true,
   "fingerprint" : "dc0f1016f495157344ac5f1296335cff725ef22f",
   "secret" : "3b935a250c50dc2c6d481d048cefdc3c",
   "timeout" : 300
 },
 "description" : "watermeter device"
}
```

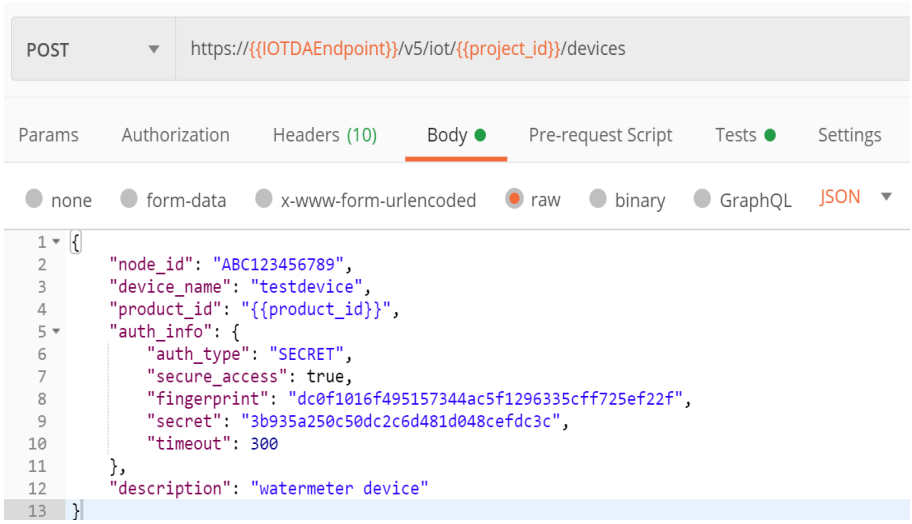
Debug the API by following the instructions provided in **Creating a Device**.

Note: Only the parameters used in the debugging example are described in the following steps.

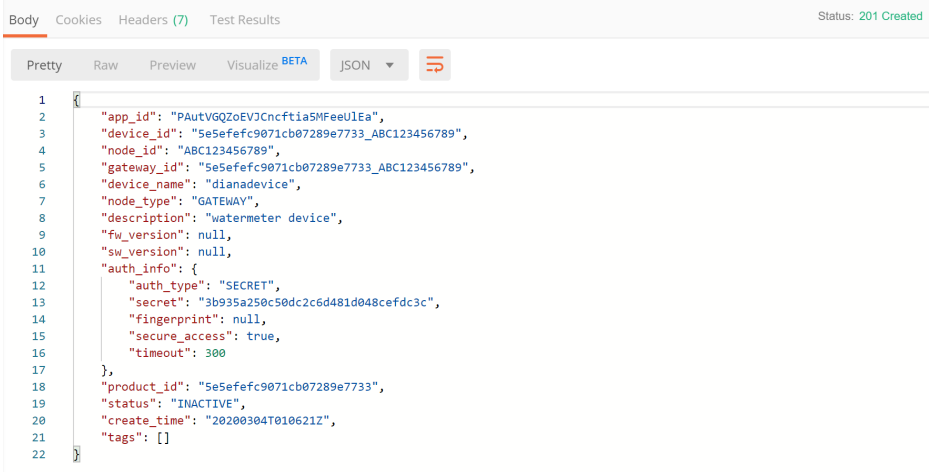
Step 1 Configure the HTTP method, URL, and headers of the API.



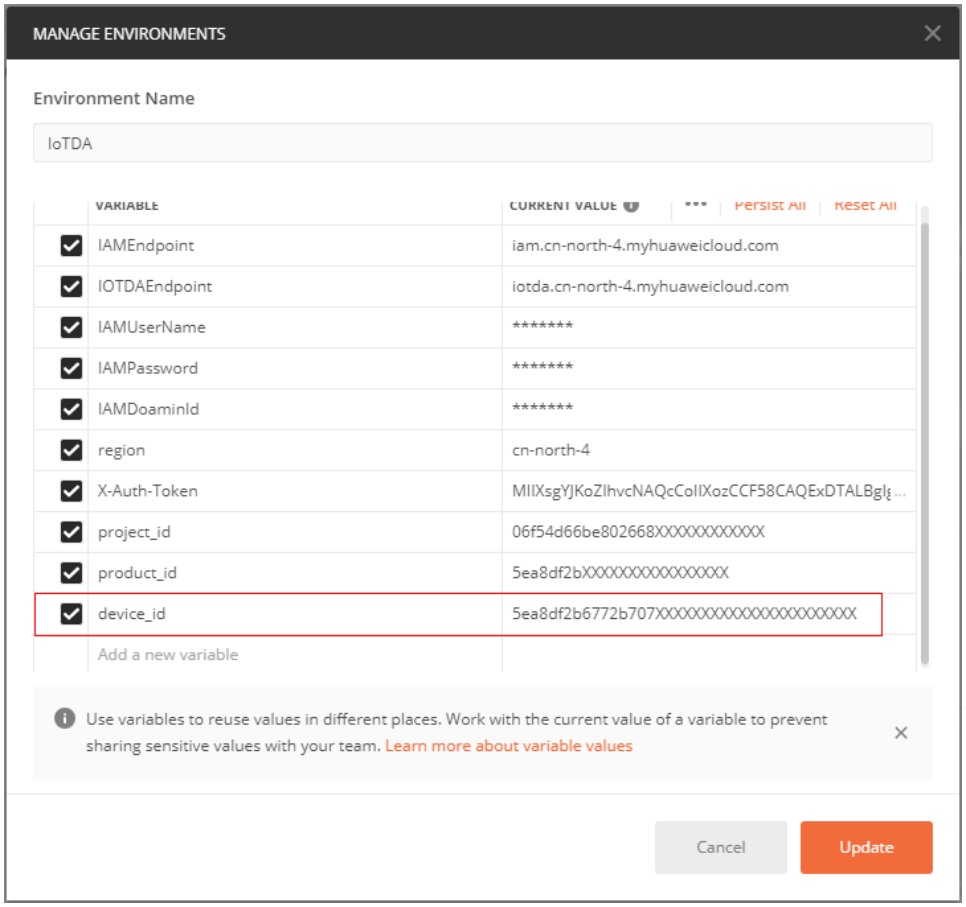
Step 2 Configure the body of the API.



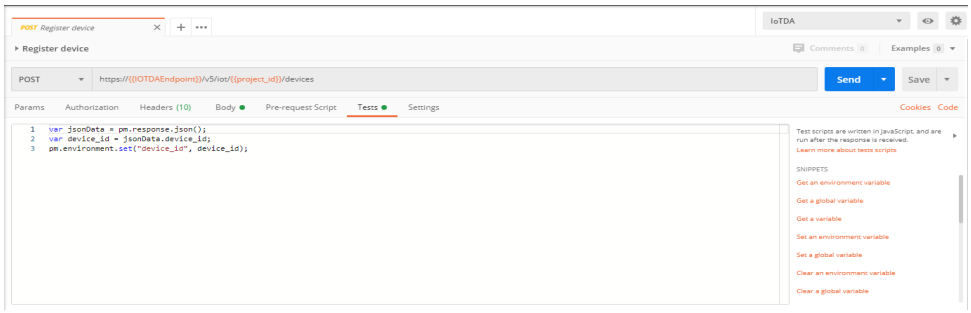
Step 3 Click **Send**. The returned code and response are displayed in the lower part of the page.



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.

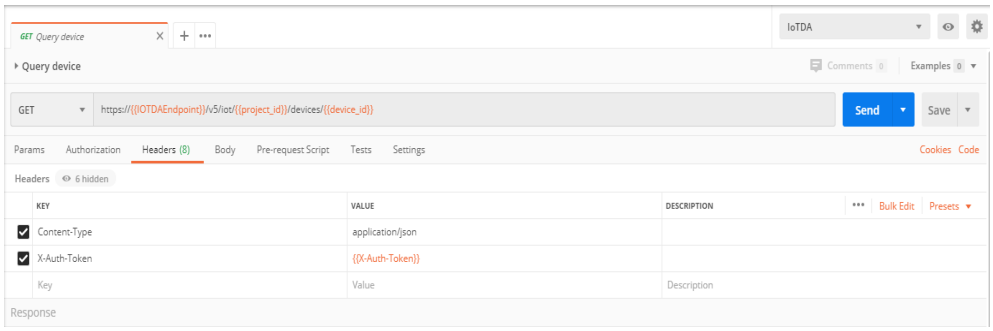
To call this API, the application constructs an HTTP request. An example request is as follows:

```
GET https://iotda.cn-north-4.myhuaweicloud.com/v5/iot/{project_id}/devices/{device_id}
Content-Type: application/json
X-Auth-Token: ********
```

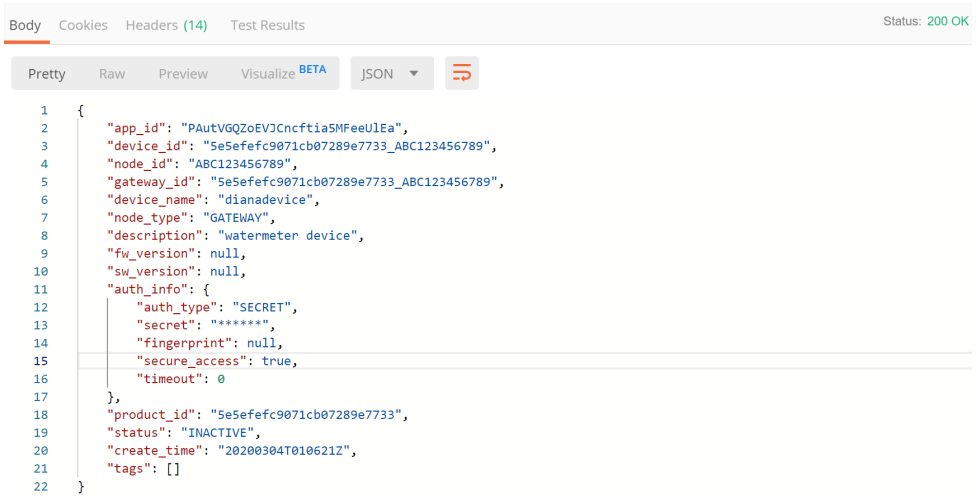
Debug the API by following the instructions provided in **Querying a Device**.

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

