IoT Device Access

Developer Guide

Issue	01
Date	2020-12-01

HUAWEI TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are trademarks 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 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 the HUAWEI CLOUD IoT platform, you must perform the operations described in the table below.

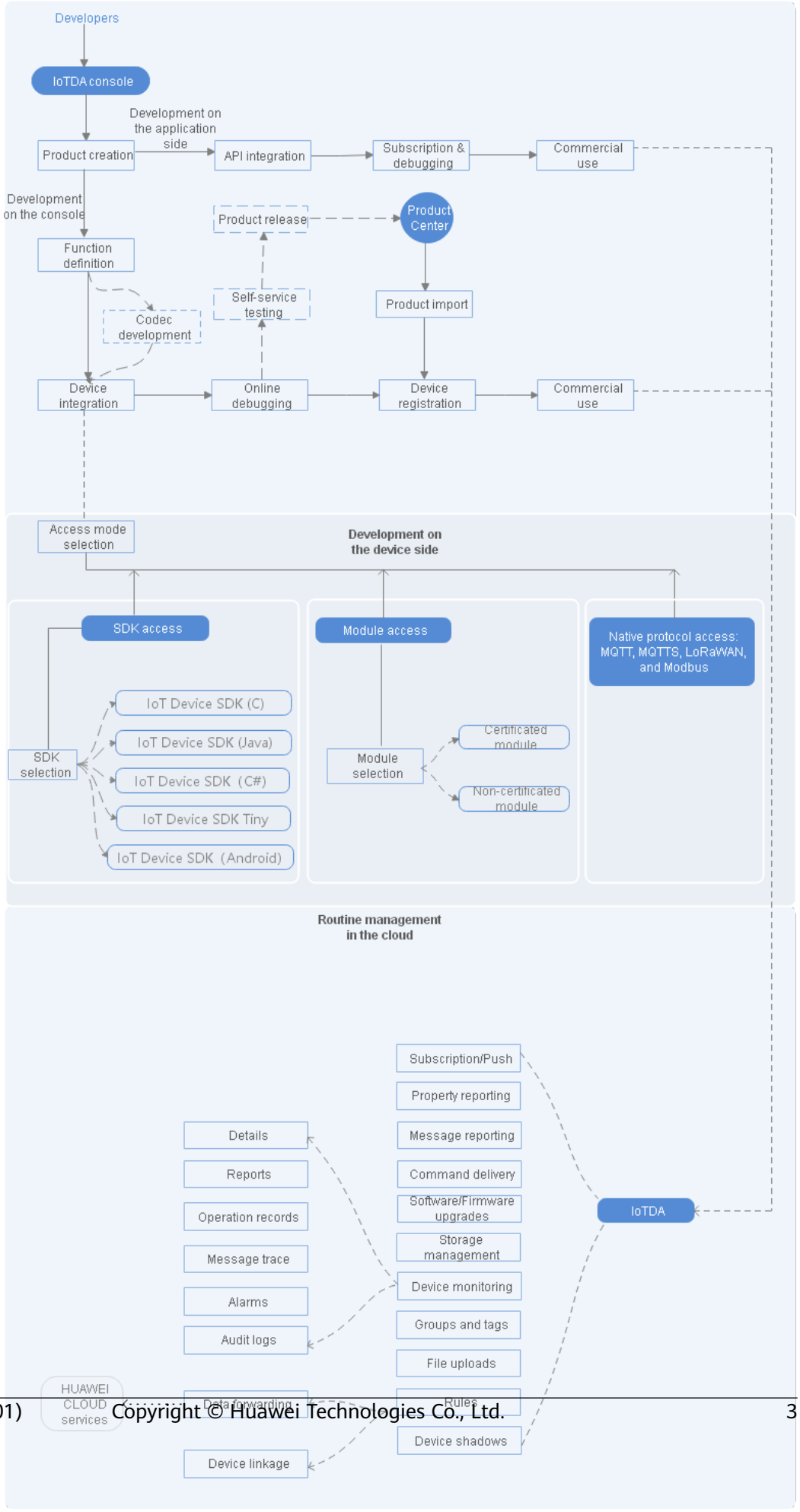
Operation	Description
Product development	Manage products, develop product models and codecs, and perform online debugging on the IoT Device Access (IoTDA) console.
Development on the application side	Carry out development for interconnection between applications and the platform, including calling APIs, obtaining service data, and managing HTTPS certificates.
Development on the device side	Carry out development for interconnection 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 figure below shows the process of using IoTDA, including product, application, device, 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 online, perform online debugging, carrying out self-service testing, and release a product. The self-service testing and product release functions are not rolled out yet.
- Application 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 development: You can connect devices to the platform by integrating SDKs or modules, or using native protocols.

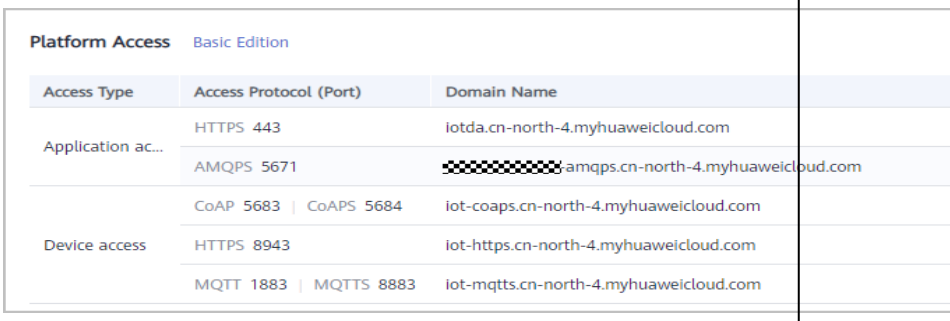
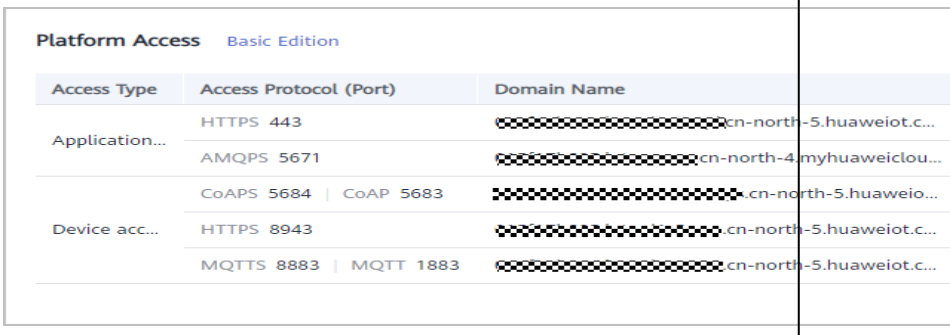
- Routine management: After a physical device is connected to the platform, you can perform routine device management on the IoTDA console or by calling APIs.

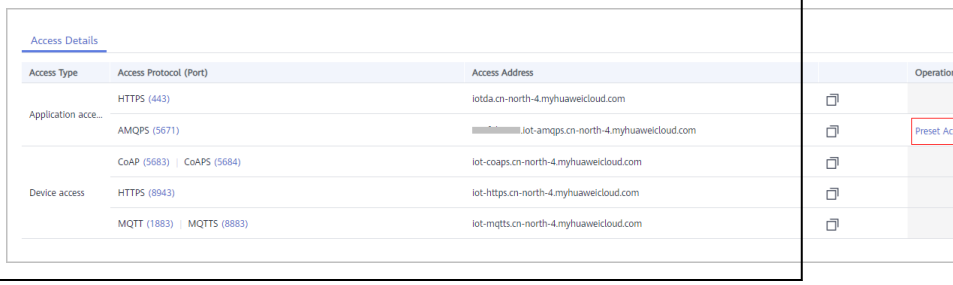


2 Obtaining Resources

Platform Connection Information

Before connecting applications and devices to the IoT platform, you must obtain platform access addresses.

Platform Environment	How to Obtain
IoT Device Access	Log in to the **IoTDA** console, access the **Overview** page, and view the device and application access addresses. **Figure 2-1** Shared domain name  For users who have subscribed to IoTDA since 00:00 on July 14, 2020 (Beijing time), the platform allocates a unique region-level ID to each user based on the shared domain name. Users can connect to the platform using their independent domain names. If you want to change a device from Basic Edition to Enterprise Edition and use a domain name for device access, you do not need to modify devices. Contact Huawei engineers to migrate data. **Figure 2-2** Independent domain names  Note: You can still use the shared domain name to access the platform.

Platform Environment	How to Obtain
	Log in to the IoTDA console, choose **IoTDA Instances** > **Basic Edition**, click **Details** to open the instance details page, and click **Preset Access Credential** to preset the accessCode and accessKey. 

Device Development Resources

The platform allows device access using MQTT or LwM2M over CoAP. Devices can connect to the platform by calling APIs or integrating with 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 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 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 sample code for calling SDK APIs. For details, see **IoT Device SDK (C#)**.	**IoT Device SDK (C#)**

Resource Package	Description	Download Link
IoT Device SDK (Android)	Devices can connect to the platform by integrating the IoT Device SDK (Android). The demo provides sample code for calling SDK APIs. For details, see **IoT Device SDK (Android)**.	**IoT Device SDK (Android)**
IoT Device SDK Tiny (C)	Devices can connect to the platform by integrating the IoT Device SDK Tiny (C). The demo provides 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 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. For details, see **Offline Codec 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 **Developing Products on the Console**.	**NB-IoT Device Simulator**
IoT Link Studio (originally named IoT Studio)	IoT Link Studio is an integrated development environment (IDE) developed for the IoT Device SDK Tiny. It provides one-stop development capabilities, such as compilation, programming, and debugging, and supports multiple development languages like C, C++, and assembly language. For details, see **Developing a Smart Street Lamp Using NB-IoT BearPi**.	**IoT Link Studio**

Application Development Resources

The platform provides a wealth of application-side APIs to ease application development. Application development is the process in which an application calls platform APIs to implement service scenarios 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. For details, see **Java Demo**.	**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**

Certificates

To connect a device to the platform in some scenarios, you must load a certificate to the device.

NOTE

This certificate applies only to the platform and must be used together with the device access domain name.

The table below describes the certificate type, format, and usage.

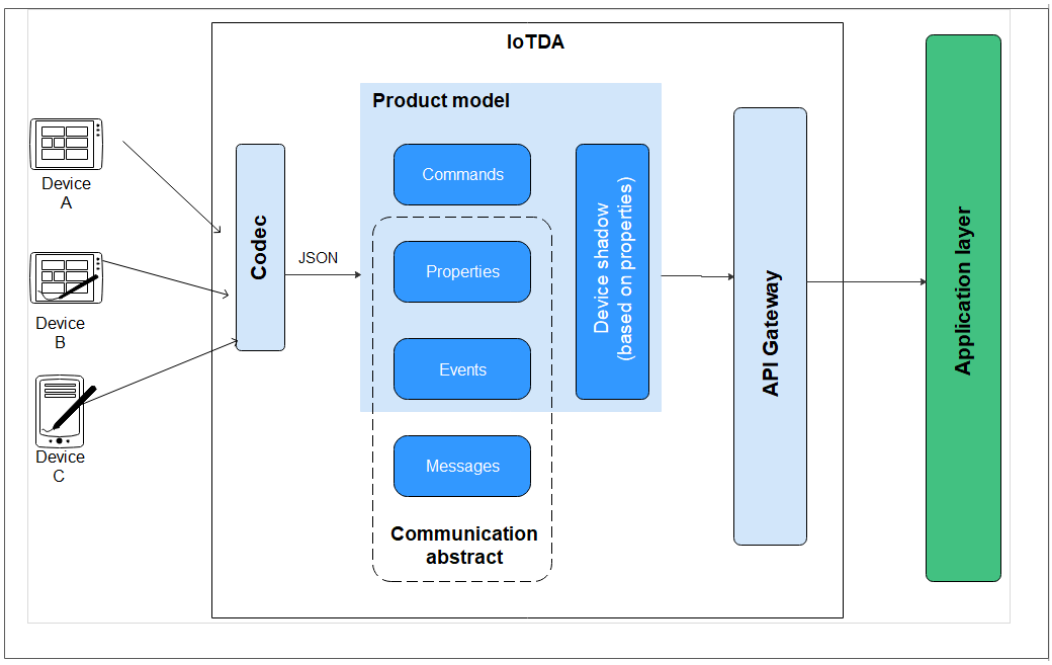
Certificate Package Name	Certificate Type	Certificate Format	Description	Download Link
certificate (Basic edition in CN North-Beijing4)	Device certificate	pem, jks, and bks	Used by a device to verify the platform identity. The certificate must be used together with the device access domain name. Note: The old domain name (iot-acc.cn-north-4.myhuaweicloud.com) must be used together with the **old certificate**.	**Certificate file**
certificate (Standard edition in CN North-Beijing4)	Device certificate	pem, jks, and bks	Used by a device to verify the platform identity. The certificate must be used together with the device access domain name.	**Certificate file**
certificate (Standard edition in CN East-Shanghai1)	Device certificate	pem, jks, and bks	Used by a device to verify the platform identity. The certificate must be used together with the device access domain name.	**Certificate file**
certificate (CN North-Beijing4)	Application certificate	pem	Used by the application to verify the platform identity in the subscription/push scenario.	**Certificate file**

3 Product Development

3.1 Product Development Guide

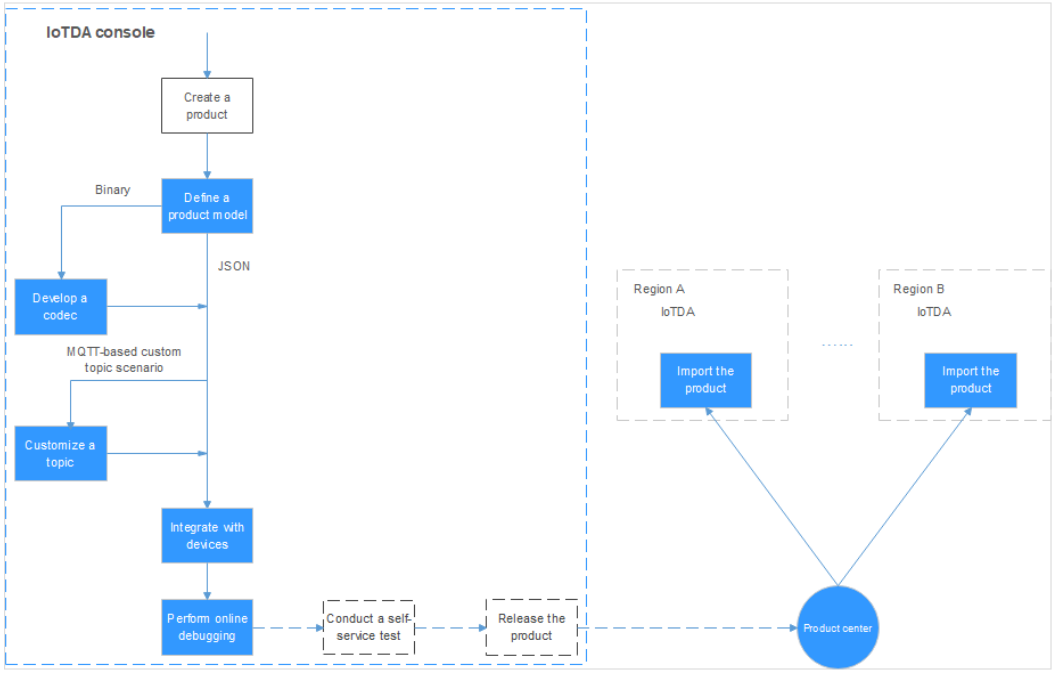
In the IoT platform integration solution, the IoT platform provides open APIs for applications to connect devices using various protocols. To provide richer device management capabilities, the IoT platform needs to understand the device capabilities and the formats of data reported by devices. Therefore, you need to develop product models and codecs to 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. Defining 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 reported data. If **Data Type** of data reported is **Binary**, a codec must be developed for the product. If **Data Type** is **JSON**, codec development is not required. The IoT platform uses codecs to convert data between the binary and JSON formats. The binary data reported by a device is decoded into the JSON format for the NA to read, and the commands delivered by the NA are encoded into the binary format for the device to understand and execute.



Product Development Process

The IoTDA console provides a one-stop development tool to help developers quickly develop products (product models and codecs) and perform self-service tests.



- 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 construct an abstract model for a device type by defining a product model on the platform, allowing the platform to understand the services, properties, and commands supported by the device.

- Codec development: If a device reports data in binary code stream format, you must develop a codec so that the platform can convert the binary format to the JSON format. If the device reports data in JSON format, you do not need to develop a codec.
- 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.

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 Log in to the **IoTDA** console.

Step 2 Click **Create Product** in the upper right corner, enter information as prompted, and click **Create** to create a product.

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 the corresponding resource space does not exist, create a **resource space** first.
Product Name	Define a product name. The product name must be unique in an account. The product name can contain letters, digits, underscores (_), and hyphens (-).
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/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. • **HTTP/HTTP2**: HTTP/HTTP2 is used by devices to access the platform. Currently, only command and property is supported. • **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.

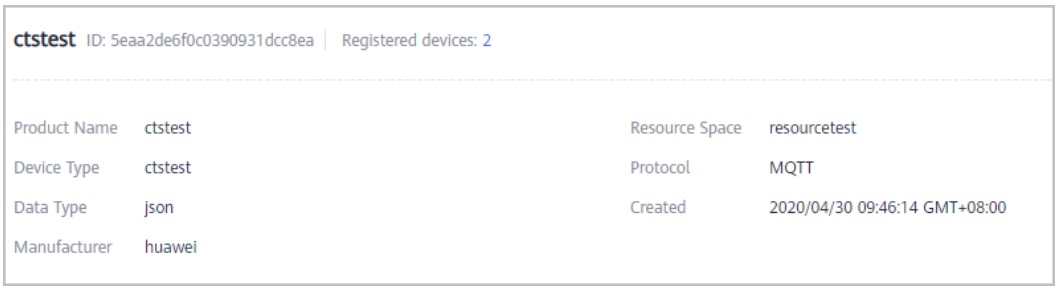
Data Type	• **JSON**: JSON is used for the communication protocol between the platform and devices. • **Binary**: You need to develop a **codec** on the IoTDA console to convert binary code data reported by devices into JSON data. The devices can communicate with the platform only after the JSON data delivered by the platform is parsed into binary code.
Manufacturer	Enter the manufacturer name of the device. The value can contain letters, digits, underscores (_), and hyphens (-).
Define Product Model	
Product Model	The platform provides multiple methods for defining a product model, such as customizing models (developing product models online), uploading models (importing product models offline), importing models from an Excel file, and importing preset models. You can select a method based on your service requirements. For details, see the following: • **Developing a Product Model Online** • **Developing a Product Model Offline** • **Exporting and Importing Product Models**
Industry	Set this parameter based on the live network environment. If the product model preset on the platform is used, set this parameter based on the industry to which the product model belongs.
Device Type	Set this parameter based on the live network environment. If the product model preset on the platform is used, the device type is automatically matched and does not need to be manually entered.

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, manufacturer name, resource space, and protocol type. The product ID is automatically generated by the platform. Other information is defined by users during **product creation**.

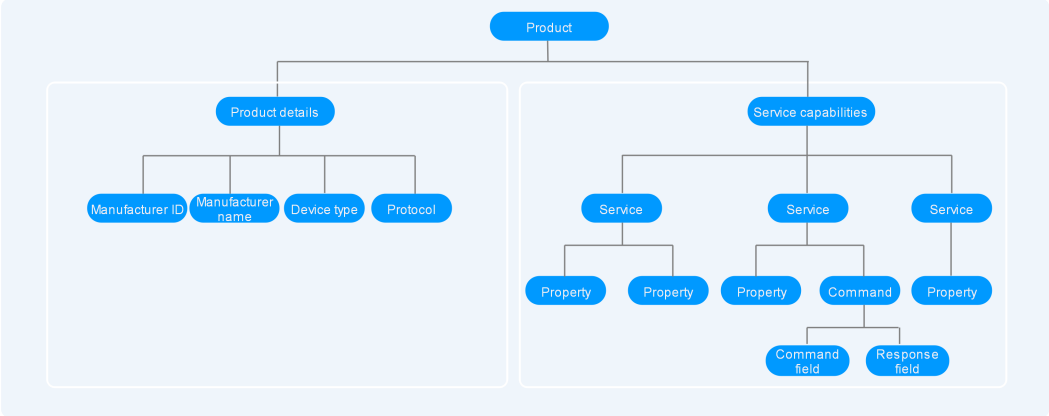


2. On the product details page, **develop a product model**, **develop a codec**, **perform online debugging**, and **customize topics**.

3.3 Developing a Product Model

3.3.1 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 and on or off switches it might have. After defining a product model, you can use it during **device registration**.



A product model consists of product details and service capabilities.

- **Product details**

  Product details describe basic information about a device, including the manufacturer ID, manufacturer name, device type, and protocol.

  For example, for a water meter, the manufacturer name could be **HZYB**, manufacturer ID **TestUtf8ManuId**, device type **WaterMeter**, and protocol **CoAP**.

- **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 profile includes five services, and each service has its own properties or commands.

Service Type	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, parameters in the commands need to be defined.
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 a water meter.
DeliverySchedule	Defines transmission rules for water meters. 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.

- **Custom Model (online development)**: Build a product model from scratch. For details, see **Developing a Product Model Online**.
- **Import Local Profile (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 Library Model**: 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 need to **create a product**. When creating a product, you need to enter information such as the product name, manufacturer name, industry, and device type. The product model uses the information as the values of device capability fields. The IoT platform provides standard models and vendor models. These models involve multiple domains and provide edited profile files. You can modify, add, or delete fields in the product model as required. If you want to custom a product model, you need to define a complete profile.

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

Procedure

Step 1 Log in to the **IoTDA** 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 **Custom Model** to add a service for the product.

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

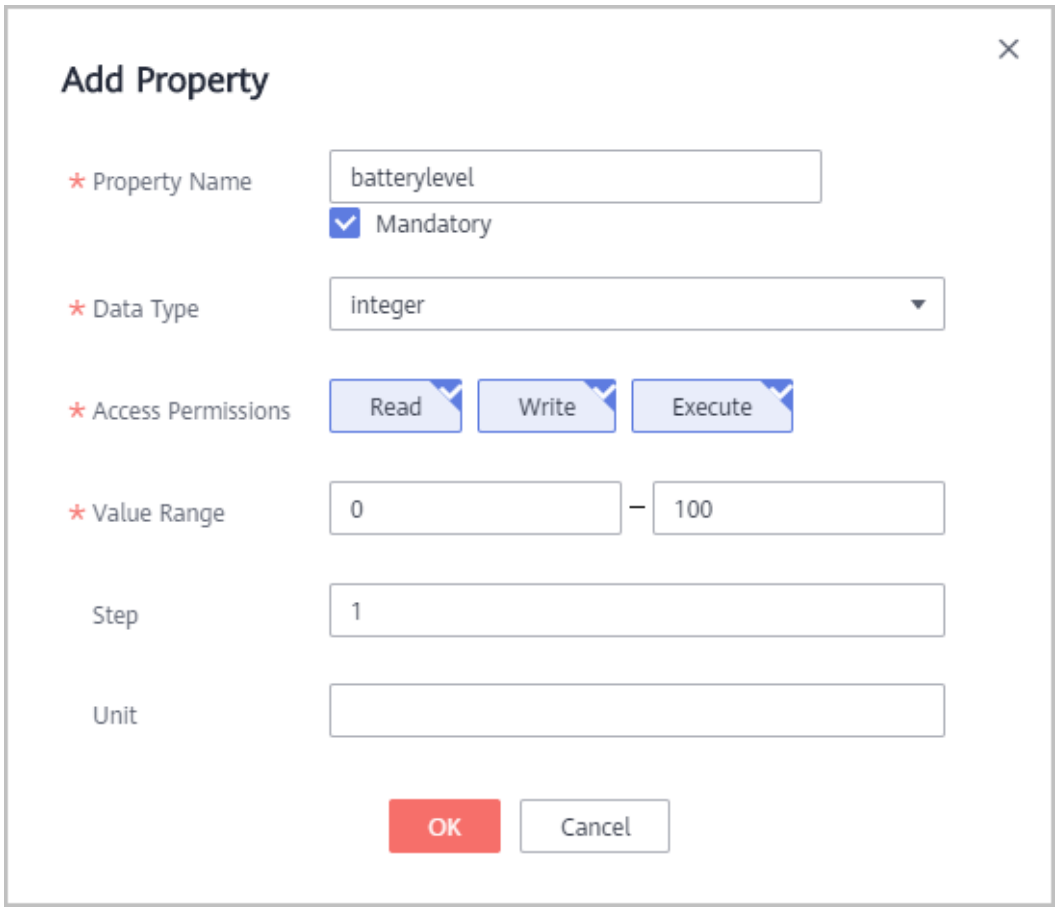
- **Service ID**: The first letter of the value must be capitalized, for example, WaterMeter and StreetLight.
- **Service Type**: You are advised to set this parameter to the service ID.
- **Description**: Define the properties of light intensity (Light_Intensity) and status (Light_Status).

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

Step 5 In the property/command list, click **Add Property**. In the dialog box displayed, set property parameters and click **OK**.

Parameter	Description
Property Name	The value of **Property Name** must start with a letter. camelCase is recommended, for example, batteryLevel and internalTemperature.
Mandatory	You are advised to select this option.

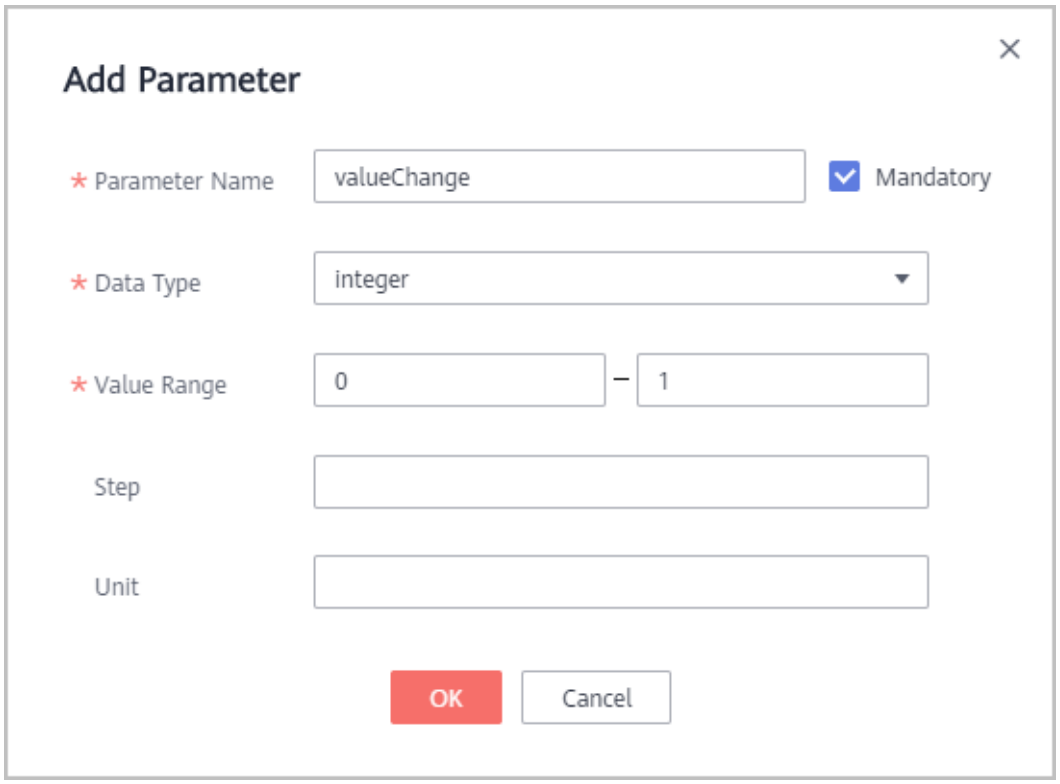
Parameter	Description
Data Type	• **int**: Select this value if the reported data is an integer or Boolean value. • **decimal**: Select this value if the reported data is a decimal. You are advised to set this parameter to **decimal** when configuring the longitude and latitude properties. • **string**: Select this value if the reported data is a string, an enumerated value, or a Boolean value. If enumerated or Boolean values are reported, use commas (,) to separate the values. • **dateTime**: Select this value if the reported data is a date. • **jsonObject**: Select this value if the reported data is in JSON structure.
Access Permissions	• **Read**: You can query the property through APIs. • **Write**: You can modify the property value through APIs. • **Execute**: After the application subscribes to the data change notification, the device reports the property value, and the application receives the push notification.
Value Range	Set these parameters according to the actual situation of the device.
Step	
Unit	



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

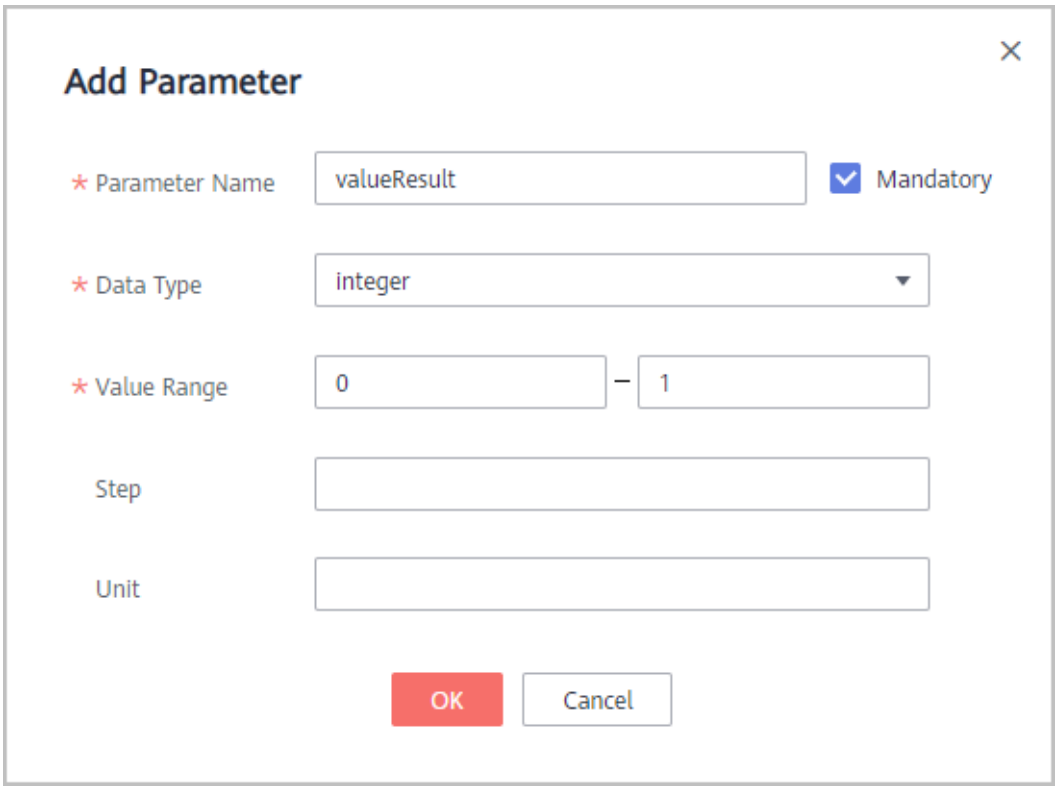
- **Command Name**: The command name must start with a letter. It is recommended that you use uppercase letters and underscores (_) to separate words, for example, DISCOVERY and CHANGE_STATUS.
- **Downlink Parameter**: Click **Add Input Parameter**. In the dialog box displayed, set the parameters of the command to be delivered and click **OK**.

Parameter	Description
Parameter Name	The parameter name must start with a letter. It is recommended that you capitalize the first letter of each word in a compound word except the first word, for example, valueChange.
Mandatory	You are advised to select this option.
Data Type	Set these parameters according to the actual situation of the device.
Value Range	
Step	
Unit	



- Click **Add Output 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	The parameter name must start with a letter. It is recommended that you capitalize the first letter of each word in a compound word except the first word, for example, valueResult.
Mandatory	You are advised to select this option.
Data Type	Set these parameters according to the actual situation of the device.
Value Range	
Step	
Unit	

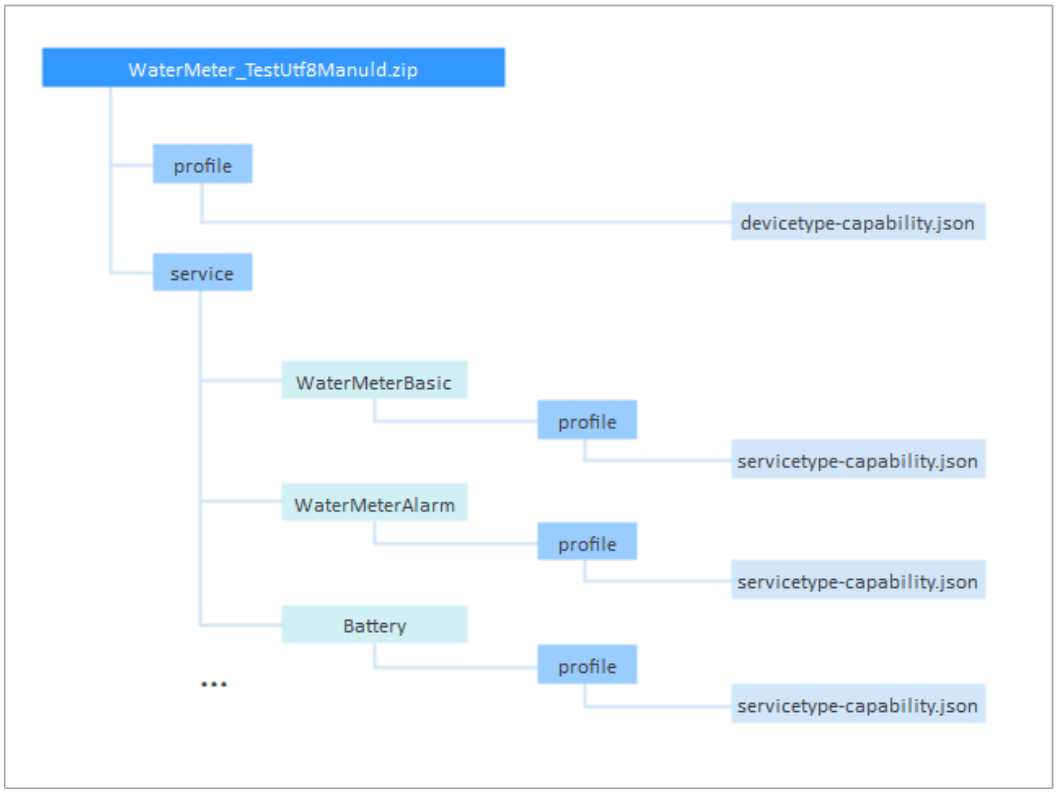


----End

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, and **WaterMeterBasic**/**WaterMeterAlarm**/**Battery** indicates the service type.



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

Therefore, **3.3.2 Developing a Product Model Online** is recommended.

Naming Rules

The profile must comply with the following naming rules:

- Capitalize device types, service types, and service IDs. Example: **WaterMeter** and **Battery**.
- Capitalize the first letter of each word in a property name except the first word, for example, **batteryLevel** and **internalTemperature**.
- For commands, capitalize all characters, with words separated by underscores. For example: **DISCOVERY** and **CHANGE_COLOR**.
- A device capability profile (.json file) must be named **devicetype-capability.json**.
- A service capability profile (.json file) must be named **servicetype-capability.json**.
- The manufacturer ID must be unique in different product models and can only be in English.
- You must ensure that 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**.

Profile Templates

To connect a new device to the IoT platform, you need to define a profile for the device. The IoT platform provides some profile 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 device profile templates, define your profile.

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 profile template provided by the IoT platform is updated continuously. The following table provides some examples of device types and service types, which are for reference only.

Device identification properties

Property	Key in the Profile	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 or Optional	Description
devices			Mandatory	Complete capability information about a device (the root node cannot be modified).
	manufacturerId		Optional	Manufacturer ID of the device.

Field	Sub-field		Mandatory or Optional	Description
	manufacturerName		Mandatory	Manufacturer name of the device (he value must be in English).
	protocolType		Mandatory	Protocol used by the device to connect to the IoT platform. For example, the value is **CoAP** for NB-IoT devices.
	deviceType		Mandatory	Type of the device.
	omCapability		Optional	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		Mandatory	Service capabilities of the device.
		serviceId	Mandatory	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	Mandatory	Type of the service. The value of this field must be the same as that of **serviceType** in the **servicetype-capability.json** file.
		option	Mandatory	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

Parameter	Sub-field	Mandatory or Optional	Description
upgradeCapability		Optional	Software upgrade capabilities of the device.
	supportUpgrade	Optional	**true**: The device supports software upgrades. **false**: The device does not support software upgrades.
	upgradeProtocolType	Optional	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		Optional	Firmware upgrade capabilities of the device.
	supportUpgrade	Optional	**true**: The device supports firmware upgrades. **false**: The device does not support firmware upgrades.
	upgradeProtocolType	Optional	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		Optional	Configuration update capabilities of the device.
	supportConfig	Optional	**true**: The device supports configuration updates. **false**: The device does not support configuration updates.
	configMethod	Optional	**file**: Configuration updates are delivered in the form of files.
	defaultConfigFile	Optional	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:

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

Parameter	Sub-field				Mandatory or Optional	Description
			dataType		Mandatory	Data type of the parameter in the command. Value: **string**, **int**, **enum**, **boolean**, , **string list**, **decimal**, **DateTime**, or **jsonObject** 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 customized JSON format, which is not parsed by the IoT platform and is transparently transmitted only.
			required		Mandatory	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		Mandatory	Minimum value. This field is valid only when **dataType** is set to **int** or **decimal**.
			max		Mandatory	Maximum value. This field is valid only when **dataType** is set to **int** or **decimal**.
			step		Mandatory	Step. This field is not used. Set it to **0**.
			maxLength		Mandatory	Character string length. This field is valid only when **dataType** is set to **string**, **string list**, or **DateTime**.

Parameter	Sub-field				Mandatory or Optional	Description
			unit		Mandatory	Unit. The value is determined by the parameter, for example: Temperature unit: C or K Percentage unit: % Pressure unit: Pa or kPa
			enumList		Mandatory	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			Mandatory	Responses to command execution.
			responseName		Mandatory	You can add _RSP to the end of **commandName** in the command corresponding to **responses**.
			paras		Mandatory	Parameters contained in a response.
				paraName	Mandatory	Name of a parameter in the command.

Parameter	Sub-field				Mandatory or Optional	Description
				dataType	Mandatory	Data type. Value: **string**, **int**, **string list**, **decimal**, **DateTime**, or **jsonObject** 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 customized JSON format, which is not parsed by the IoT platform and is transparently transmitted only.
				required	Mandatory	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.
				min	Mandatory	Minimum value. This field is valid only when **dataType** is set to **int** or **decimal**. The value must be greater than or equal to the value of **min**.
				max	Mandatory	Maximum value. This field is valid only when **dataType** is set to **int** or **decimal**. The value must be less than or equal to the value of **max**.
				step	Mandatory	Step. This field is not used. Set it to **0**.
				maxLength	Mandatory	Character string length. This field is valid only when **dataType** is set to **string**, **string list**, or **DateTime**.

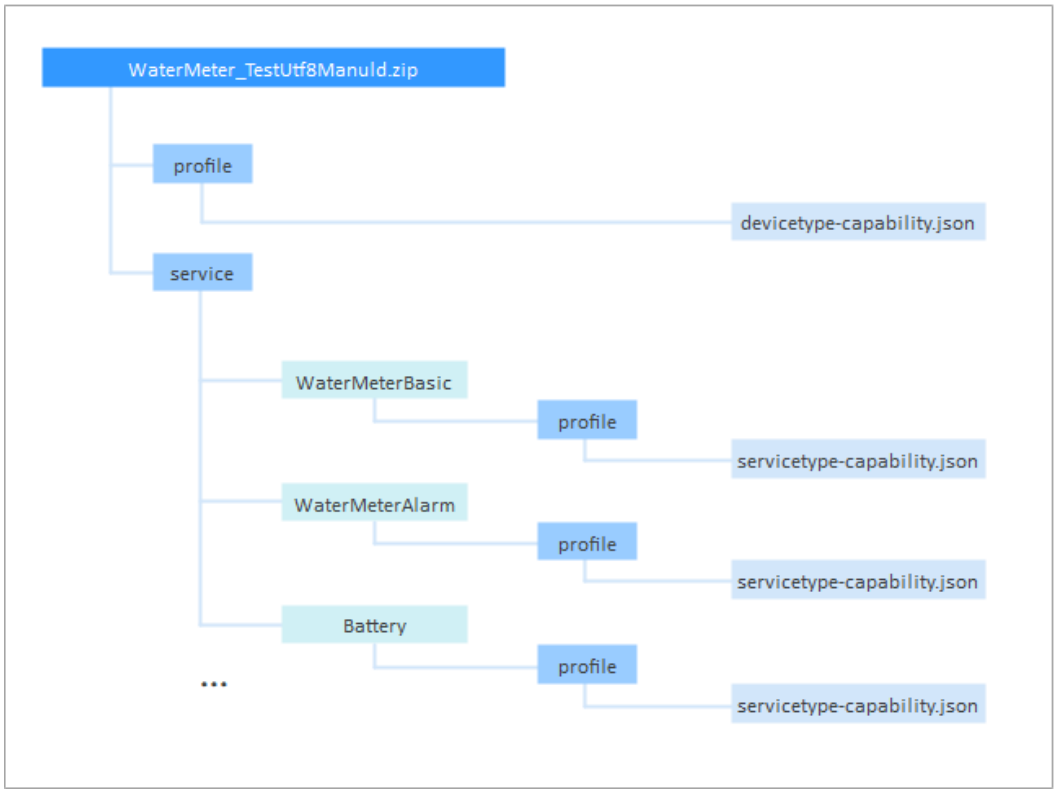
Parameter	Sub-field				Mandatory or Optional	Description
				unit	Mandatory	Unit. The value is determined by the parameter, for example: Temperature unit: C or K Percentage unit: % Pressure unit: Pa or kPa
				enumList	Mandatory	List of enumerated values. For example, the status of a switch can be set 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				Mandatory	Reported data. Each sub-node indicates a property.
		propertyName			Mandatory	Name of the property.
		dataType			Mandatory	Data type. Value: **string**, **int**, **string list**, **decimal**, **DateTime**, or **jsonObject** 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 customized JSON format, which is not parsed by the IoT platform and is transparently transmitted only.

Parameter	Sub-field				Mandatory or Optional	Description
		required			Mandatory	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			Mandatory	Minimum value. This field is valid only when **dataType** is set to **int** or **decimal**. The value must be greater than or equal to the value of **min**.
		max			Mandatory	Maximum value. This field is valid only when **dataType** is set to **int** or **decimal**. The value must be less than or equal to the value of **max**.
		step			Mandatory	Step. This field is not used. Set it to **0**.
		method			Mandatory	Access mode. **R** indicates reading, **W** indicates writing, and **E** indicates subscription. Value: R, RW, RE, RWE, or null
		unit			Mandatory	Unit. The value is determined by the parameter, for example: Temperature unit: C or K Percentage unit: % Pressure unit: Pa or kPa
		maxLength			Mandatory	Character string length. This field is valid only when **dataType** is set to **string**, **string list**, or **DateTime**.

Parameter	Sub-field				Mandatory or Optional	Description
		enumList			Mandatory	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 profile 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 names in orange cannot be changed.

- The product model is compressed in **.zip** format.
- The product model must be named in the format of deviceType_manufacturerId. The values of **deviceType**, **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 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 Product Models

Product models 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 edit a product model, you can directly import the product model to the platform.

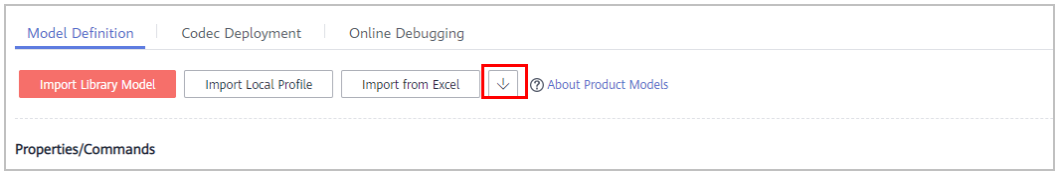
Exporting a Product Model

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

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

Step 2 In the navigation pane, choose **Products**. In the product list, select a product and click **View**.

Step 3 On the product details page, click  to download the product model to the local host.



----End

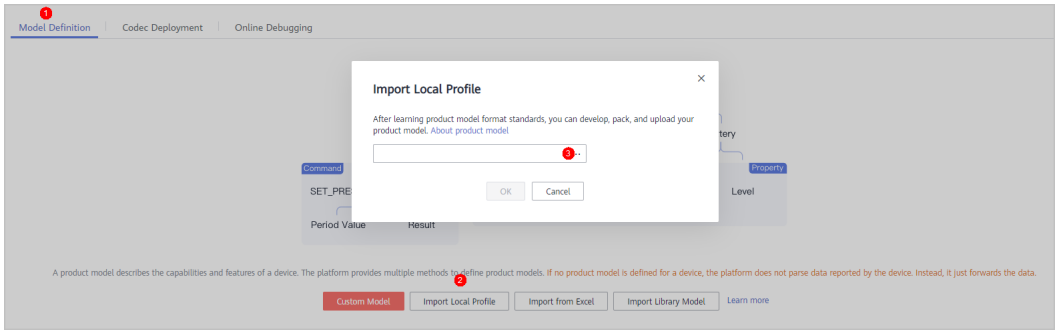
Importing a Product Model

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

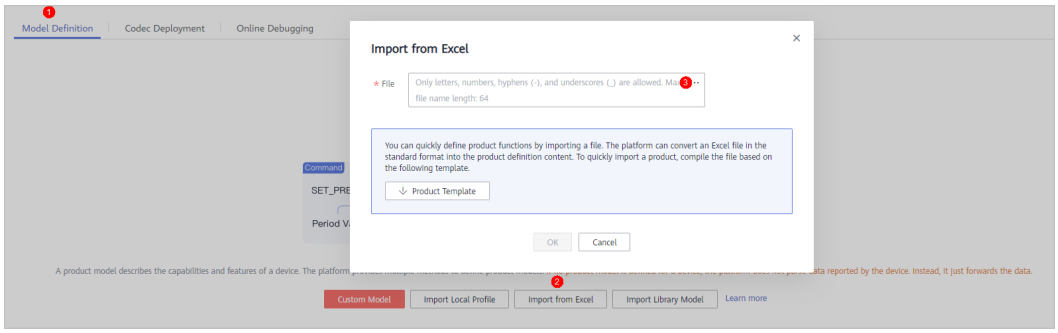
NOTE

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 Local Profile**
  a. Log in to the IoTDA console.
  b. In the navigation pane, choose **Products**. In the product list, select a product and click **View**.
  c. On the **Model Definition** tab page, click **Import Local Profile**. In the dialog box displayed, load the local profile and click **OK**.



- **Import from Excel**
  a. Log in to the IoTDA console.
  b. In the navigation pane, choose **Products**. In the product list, select a product and click **View**.
  c. On the **Model Definition** tab page, click **Import from Excel**. In the product template downloaded, enter the service ID on the **Device** sheet and set parameters such as properties, commands, and events on the **Parameter** sheet. Import the Excel file and click **OK**.

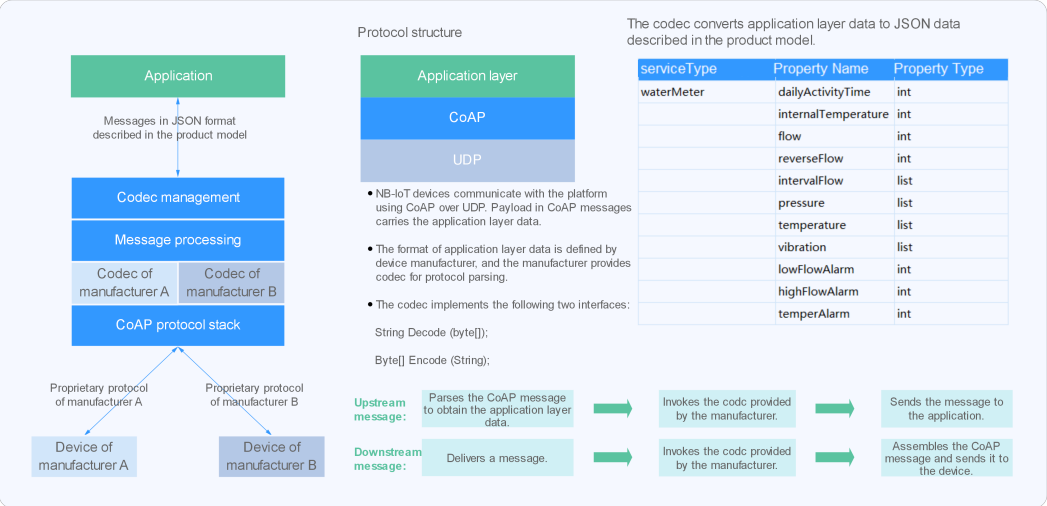


3.4 Developing a Codec

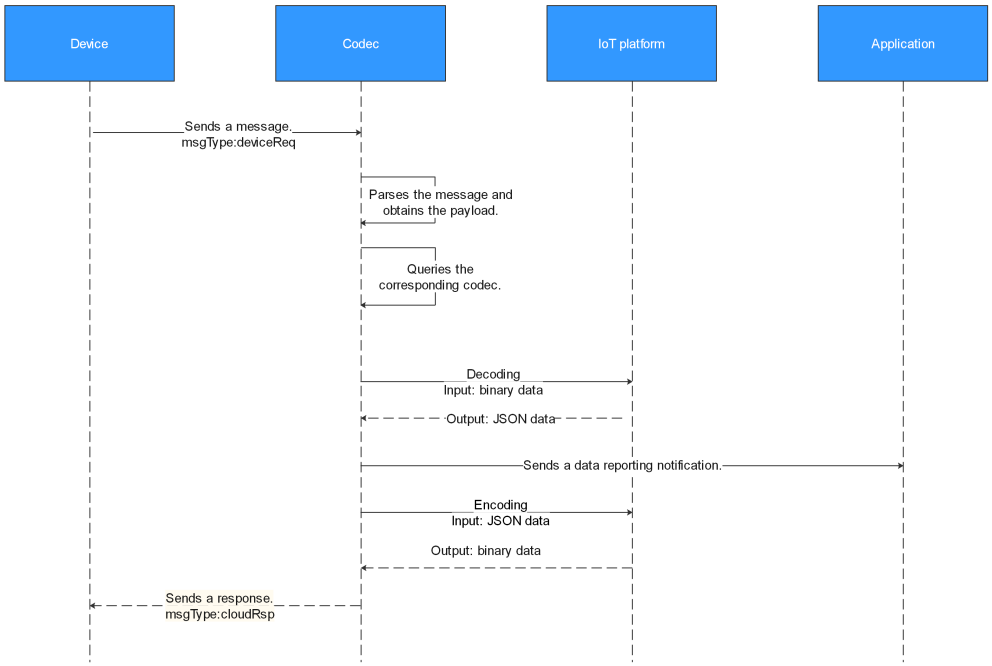
3.4.1 Definition

If a device reports binary data, a codec must be developed for data format conversion. If a device reports JSON data, codec development is not required.

For example, in the NB-IoT scenario where devices communicate with the IoT platform using CoAP, the payload of the CoAP message is data at the application layer and the data type is defined by the device. As NB-IoT devices require low power consumption, data at the application layer is 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.



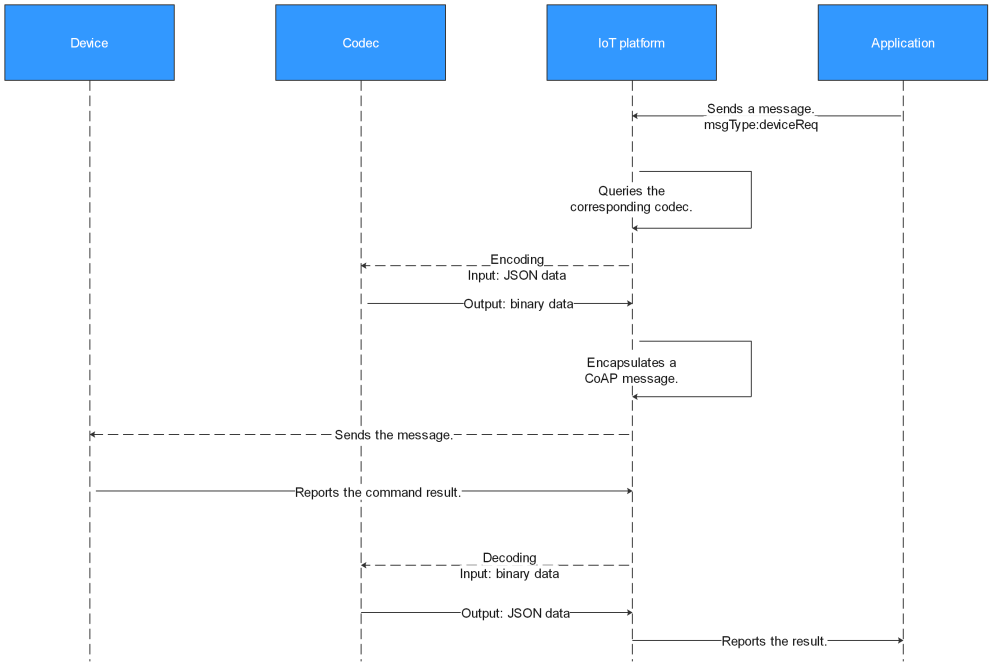
Data Reporting



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 and sending the encoded data to a device

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 Offline Development

The platform provides three methods for developing codecs. Offline codec development is complex and time-consuming. Graphical codec development is recommended.

- **Graphical development**: The codec of a product can be quickly developed in a visualized manner on the IoTDA console.
- **Offline development**: A codec is developed through the secondary development based on the Java codec demo to implement encoding, decoding, packaging, and quality inspection.
- **Script-based development**: JavaScript scripts are used to implement encoding and decoding.

3.4.2 Graphical Development

Currently, Huawei IoT platform codecs are developed only for NB-IoT devices.

On the IoTDA console, you can quickly develop codecs in a visualized manner. Some preset product models contain developed codecs. If you use such a product model to create a product, you can directly use or modify the codec. If you choose to customize a product, you need to develop a codec.

This section uses an NB-IoT smoke detector as an example to describe how to develop an 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

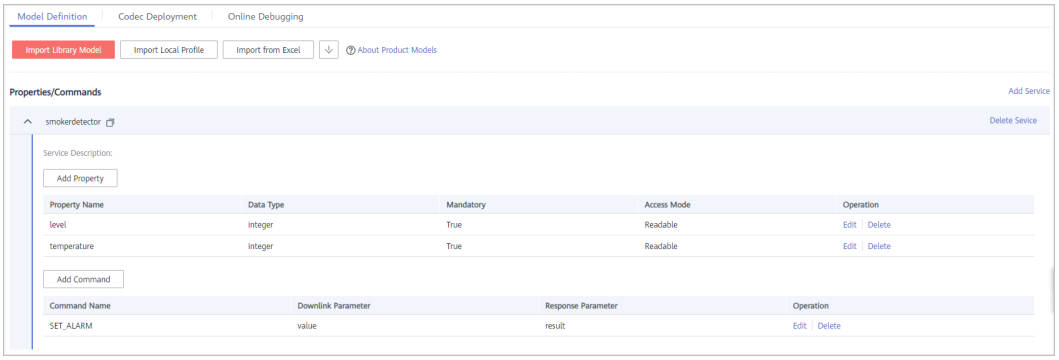
A smoke detector provides the following functions:

- Reporting smoke alarms (fire severity) and temperature
- Remote control commands, which can 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 is disabled, and the value **1** indicates that the alarm is enabled.

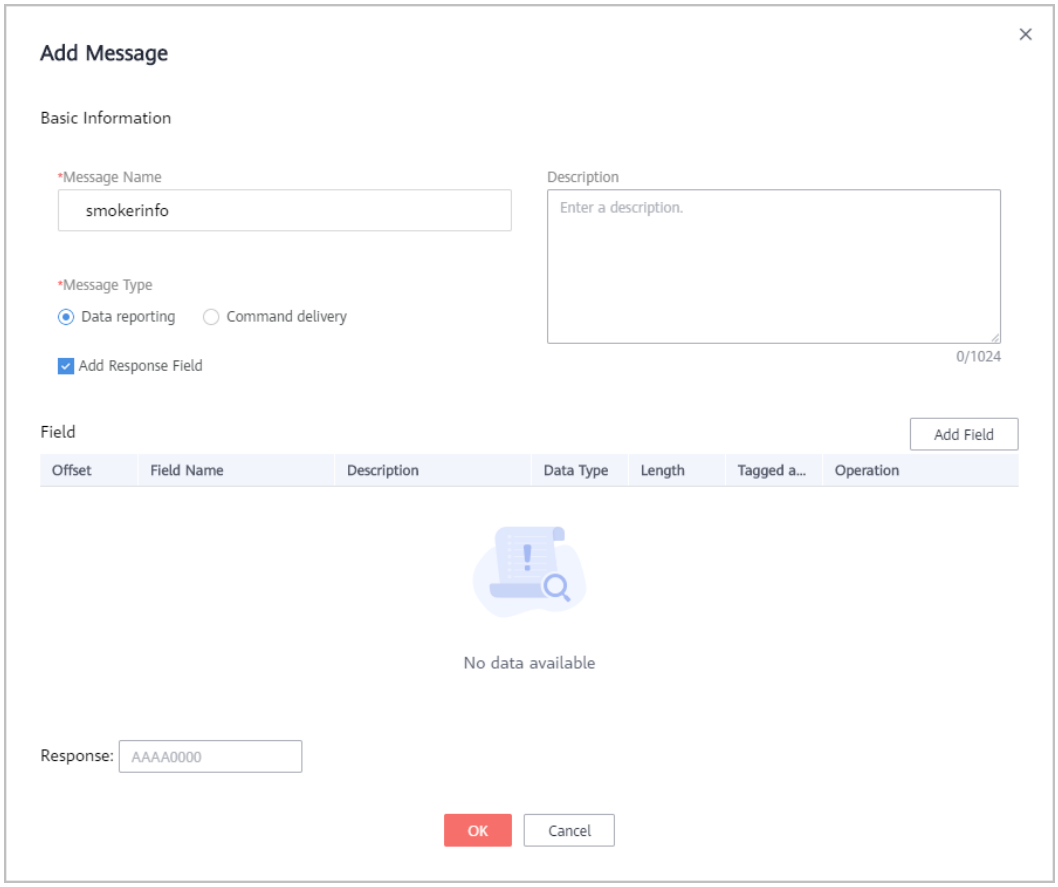


Developing a Codec

Step 1 On the product details page of the smoke detector, select **Codec Development** and click **Online Develop**.

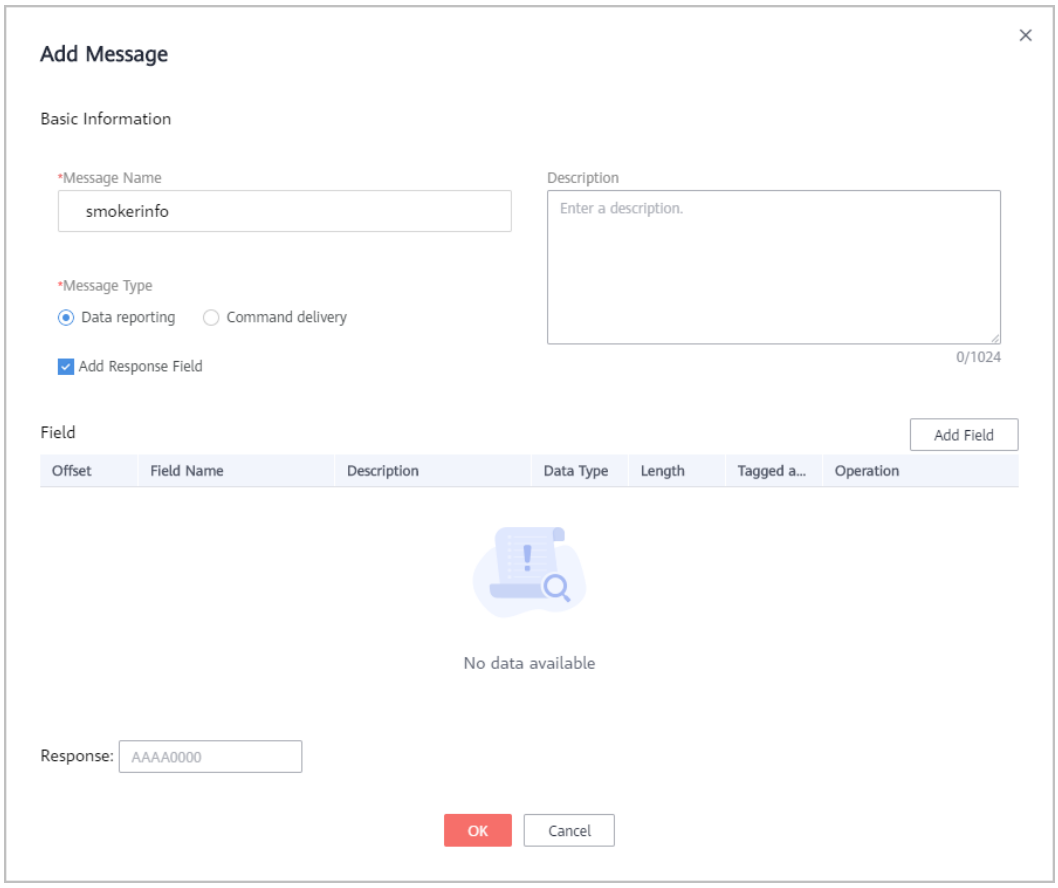
Step 2 Click **Add Message** to add a **smokerinfo** message. This step is performed to decode the binary code stream message uploaded by the device to the JSON format so that the platform can understand the message. The following is a configuration example:

- **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)

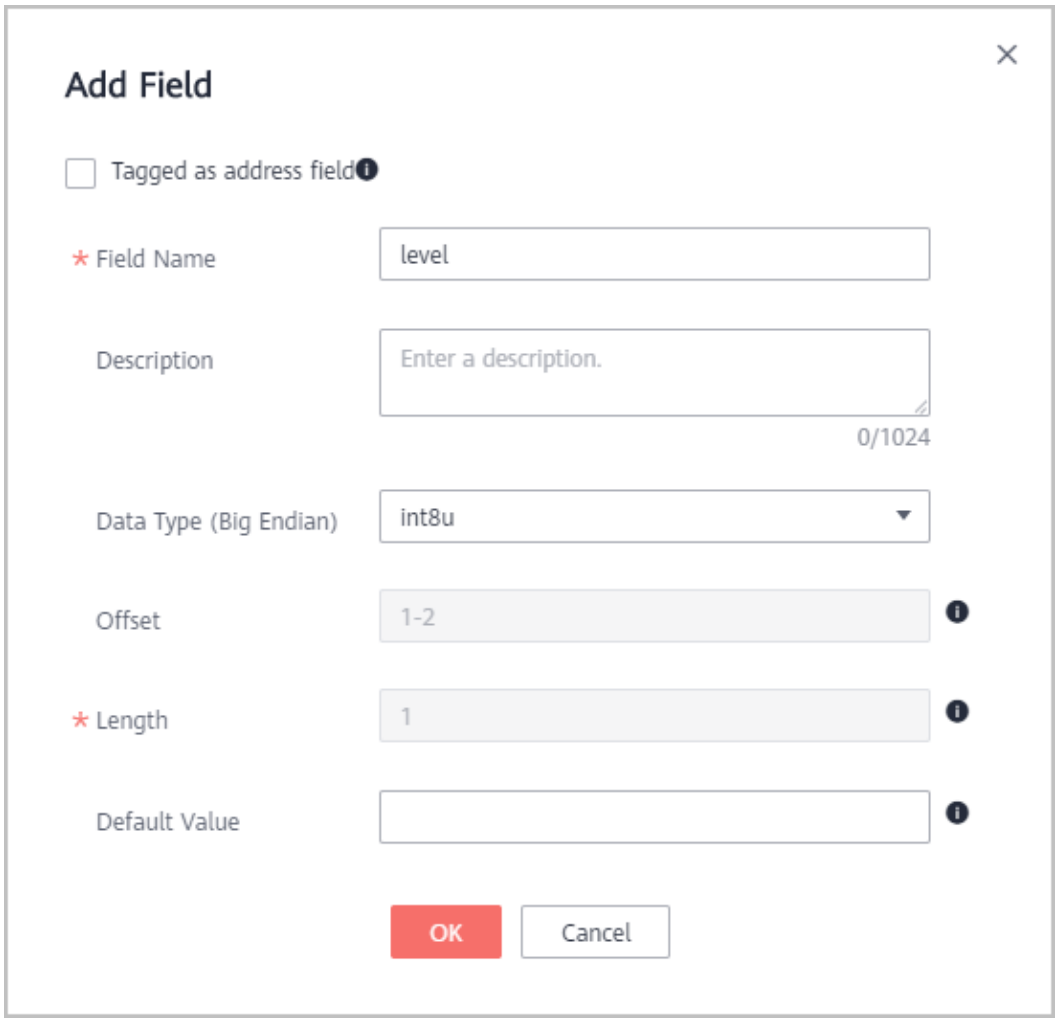


1. Click **Add Field**, select **Tagged as address field**, and add the **messageID** field, which indicates the message type. In this scenario, the message type for reporting the fire severity and temperature is 0x0. When a device reports a message, the first field of each message is **messageID**. 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**.

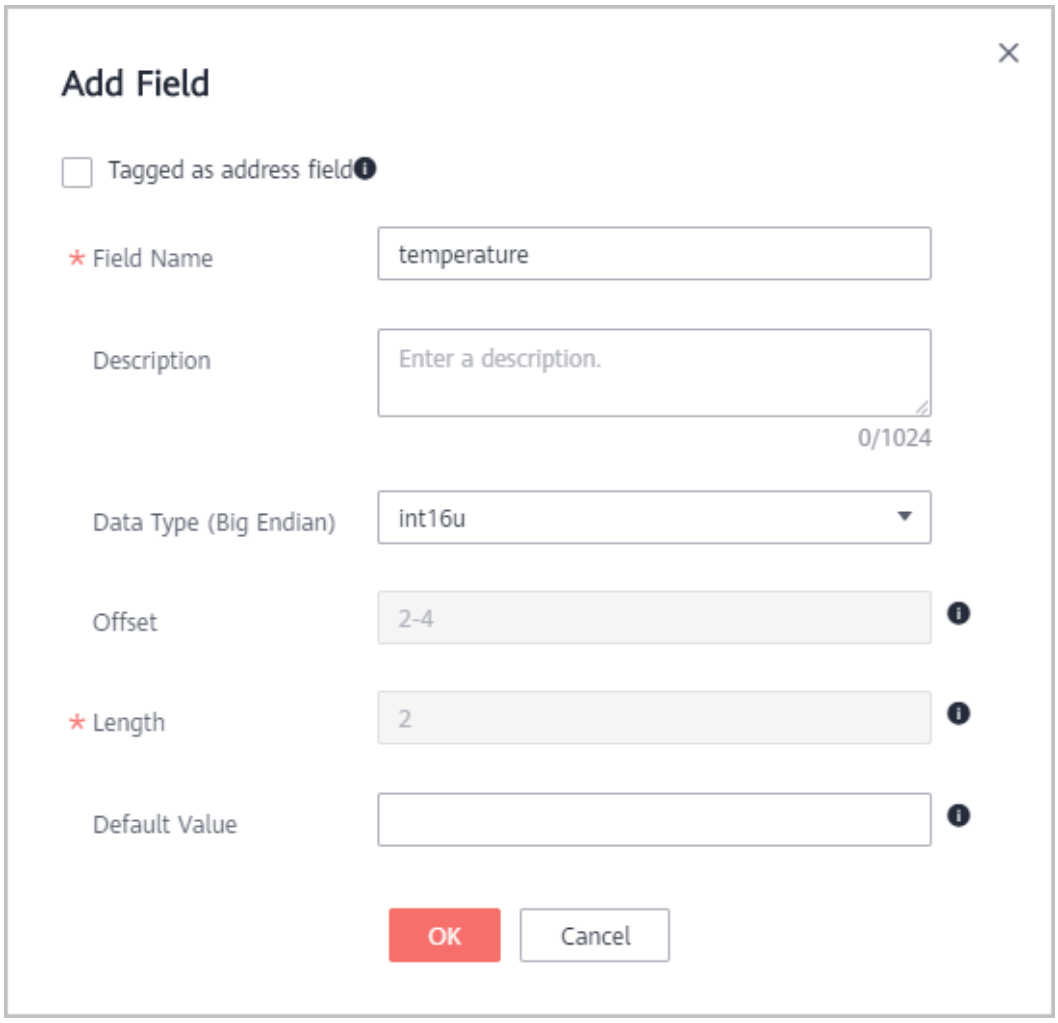
- **Default Value** can be changed but must be in hexadecimal format. In addition, the corresponding field in data reporting messages must be the same as the default value.



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, set **Data Type** to **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, set **Offset** to **1-2**.
   - The value of **Length** is automatically filled based on **Data Type**.
   - If you do not set **Default Value**, the value of temperature is not fixed and has no default value.

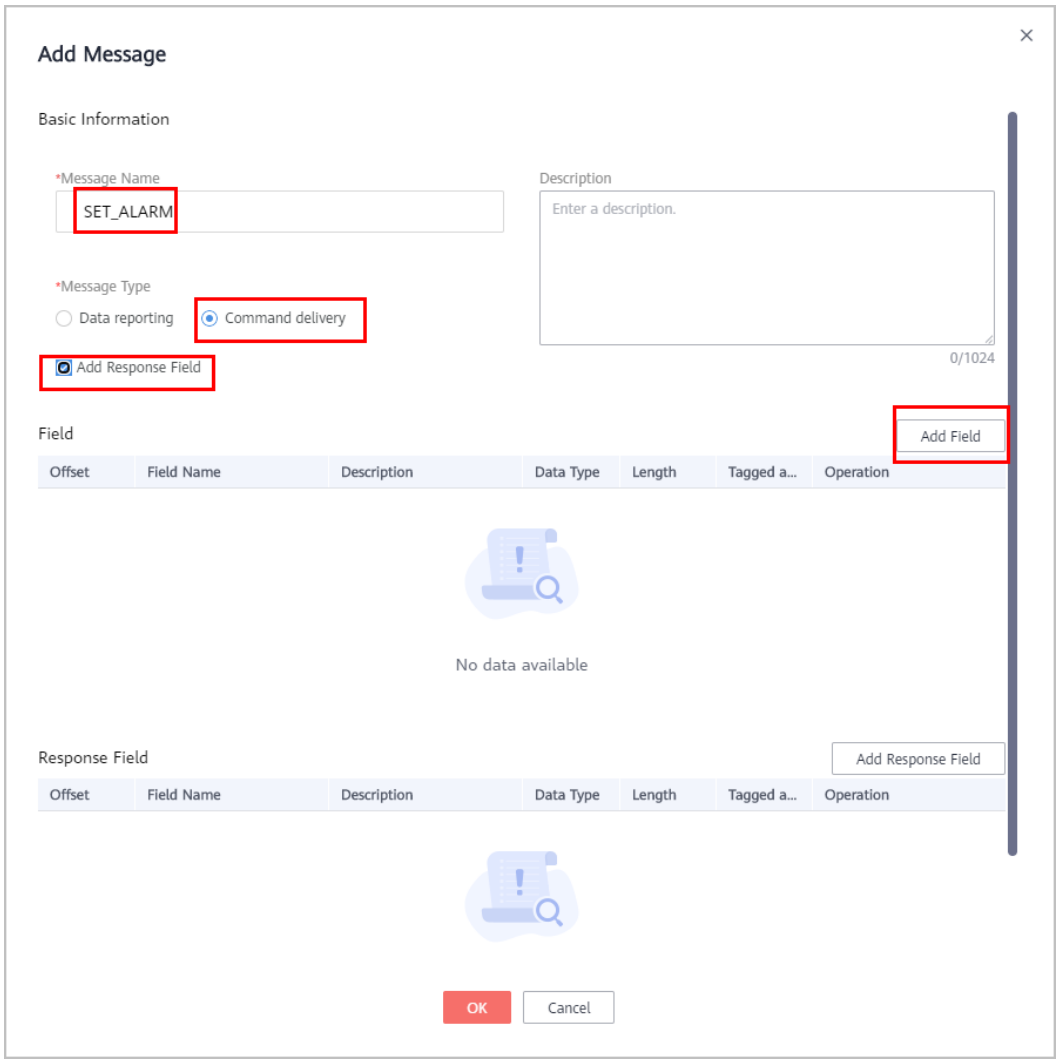


3. Add the **temperature** field to indicate the temperature at the fire scene.
   - **Data Type**: In the product model, the data type of the **temperature** property is **int** and the maximum value is **1000**. Therefore, set **Data Type** to **int16u** in the codec to meet the value range of the **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, set **Offset** to **2-4**.
   - The value of **Length** is automatically filled based on **Data Type**.
   - If you do not set **Default Value**, the value of temperature is not fixed and has no default value.

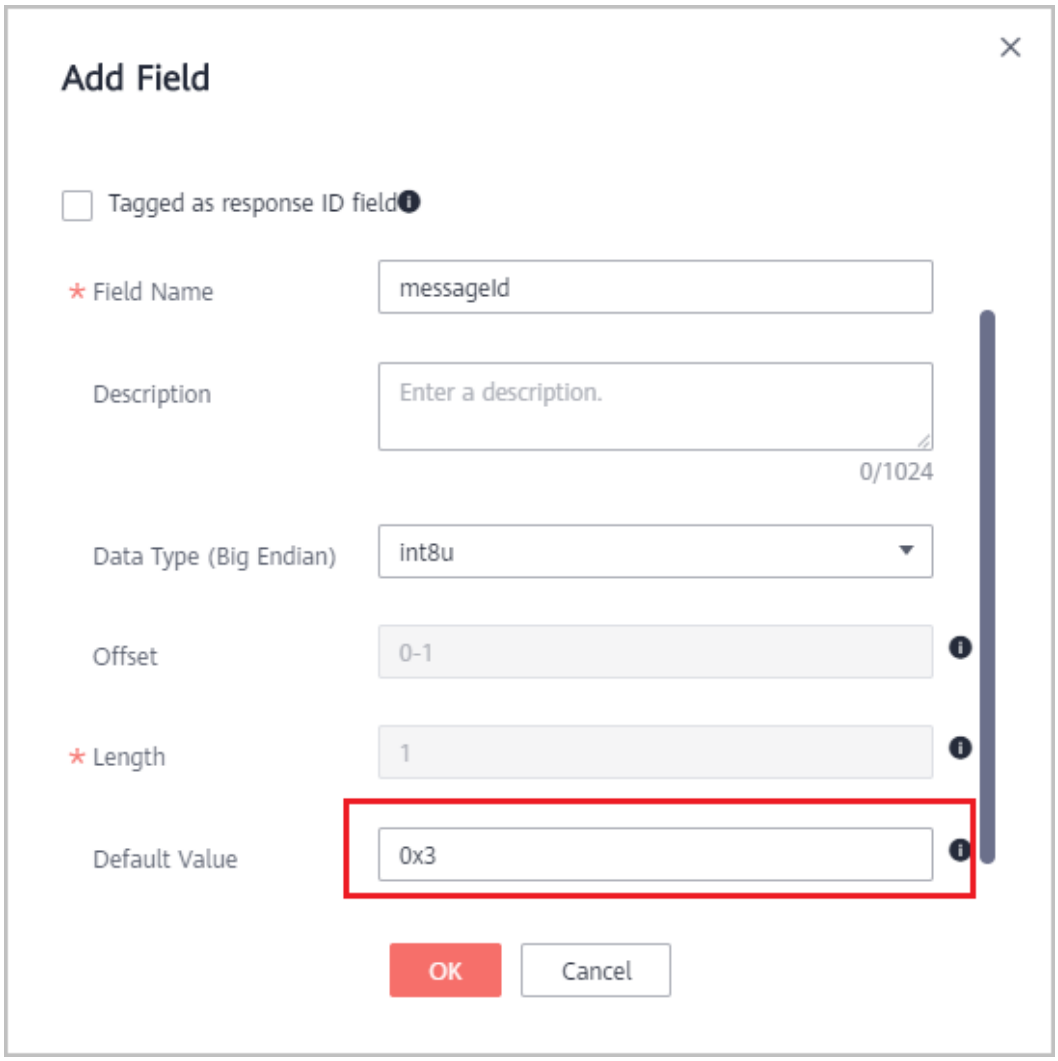


Step 3 Click **Add Message** to add a SET_ALARM message and set the temperature threshold for fire alarms. For example, if the temperature exceeds 60°C, the device reports an alarm. This step is performed to encode the command message in JSON format delivered by the IoT platform into binary data so that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

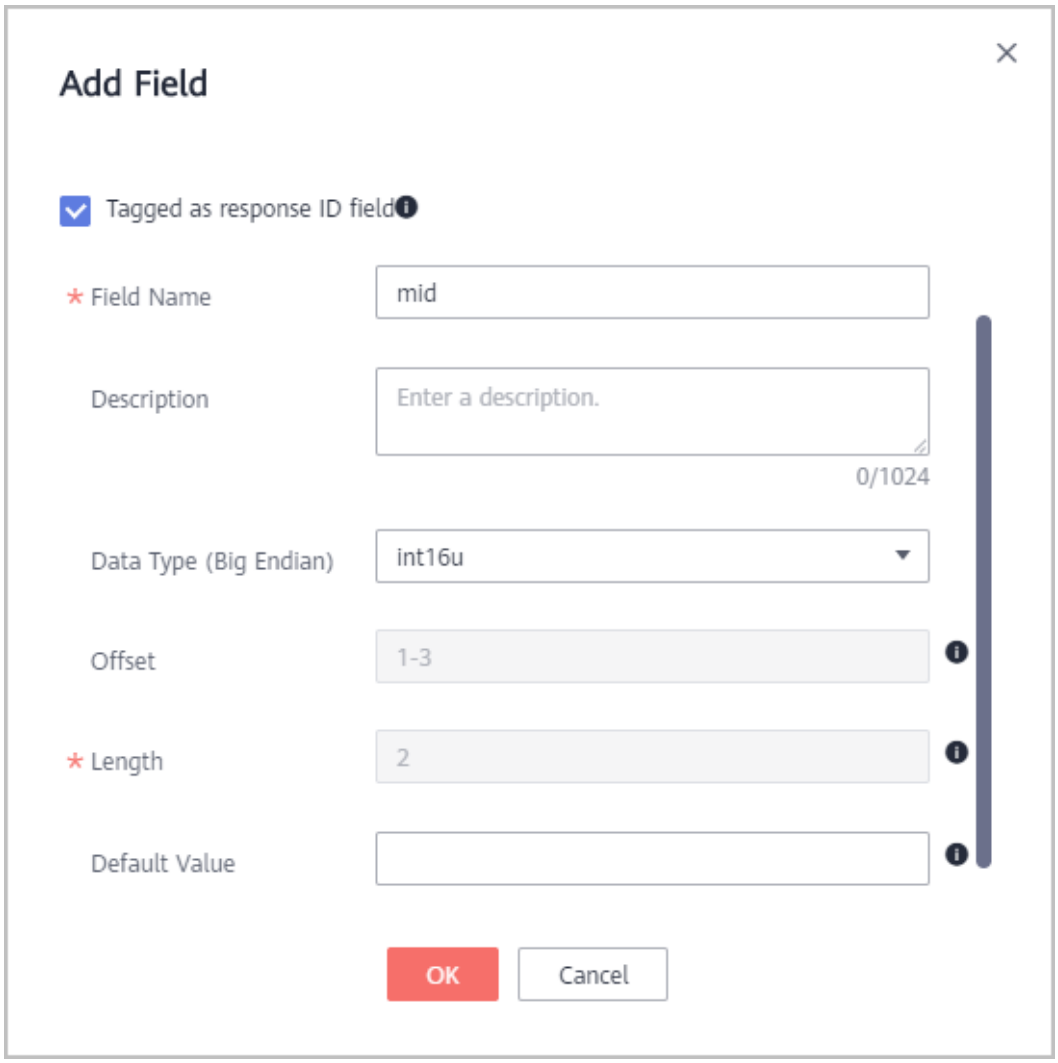
- Message Name: SET_ALARM
- Message Type: Command delivery
- Add Response Field: selected. After a response field is added, the device reports the command execution result after receiving the command. You can determine whether to add response fields as required.



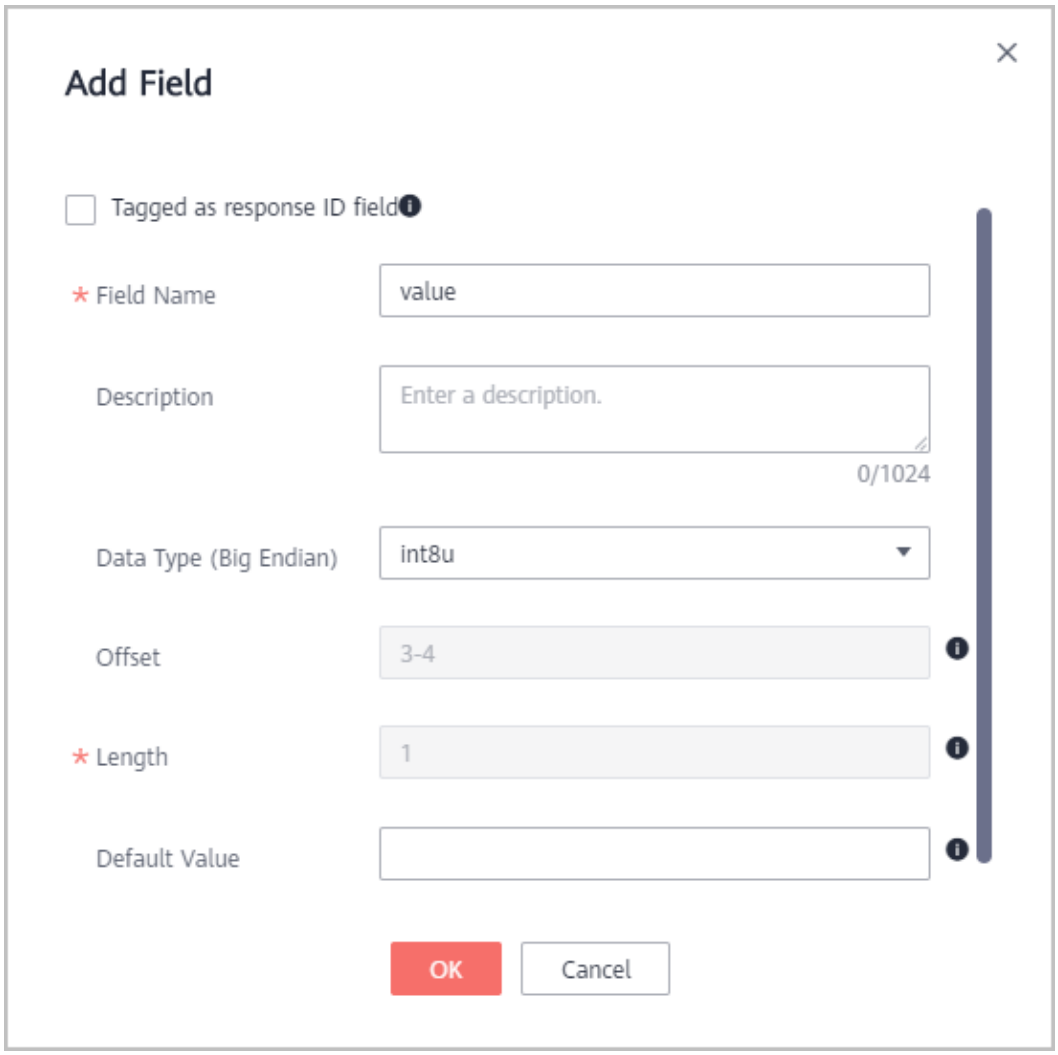
a. Click **Add Field** to add the **messageID** field, which indicates the message type. For example, set the message type of the fire alarm threshold to **0x3**. For details about the messageID, data type, length, default value, and offset, see **1**.



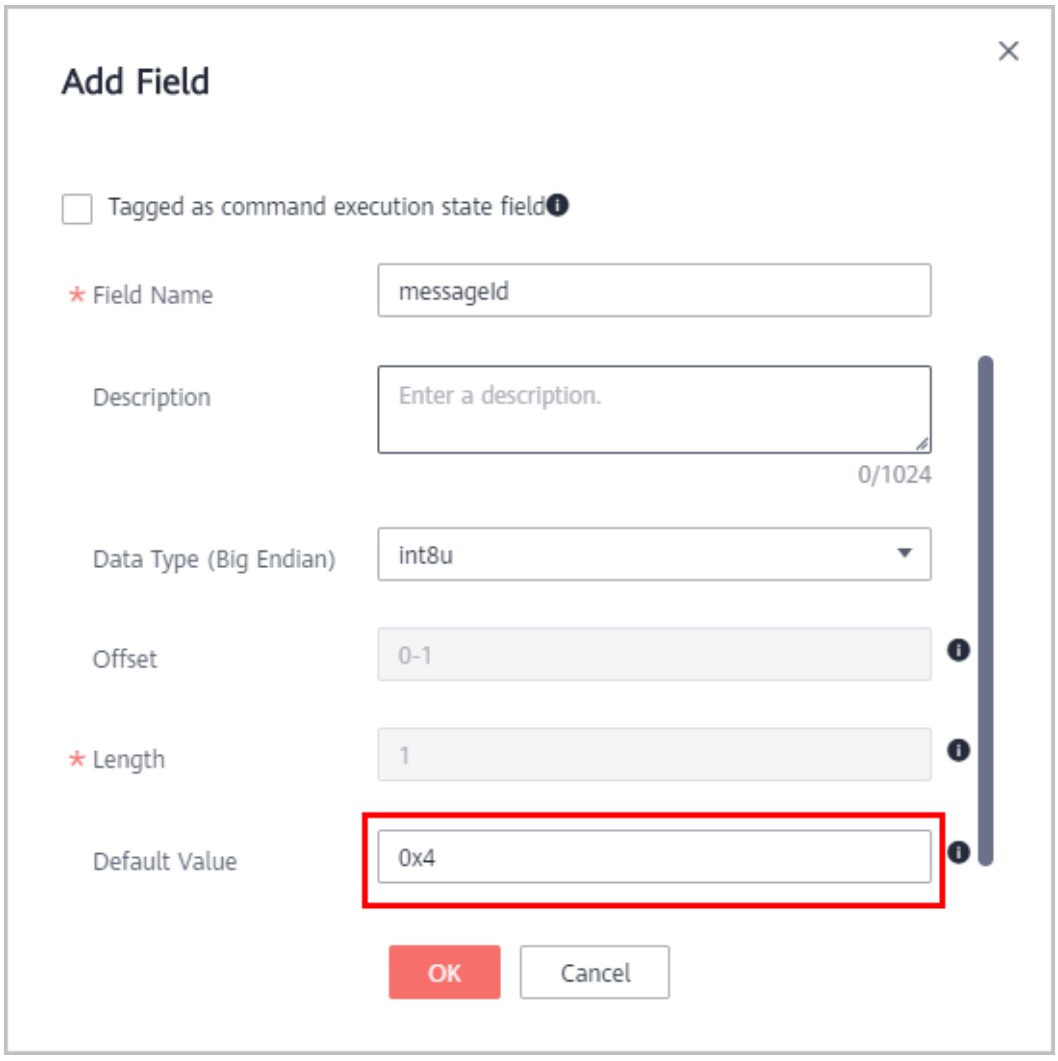
b. Add the **mid** field. This field is generated and delivered by the platform and is used to associate the delivered command with the command delivery response. The data type of the **mid** field is **int16u** by default. For details about the length, default value, and offset, see **2**.



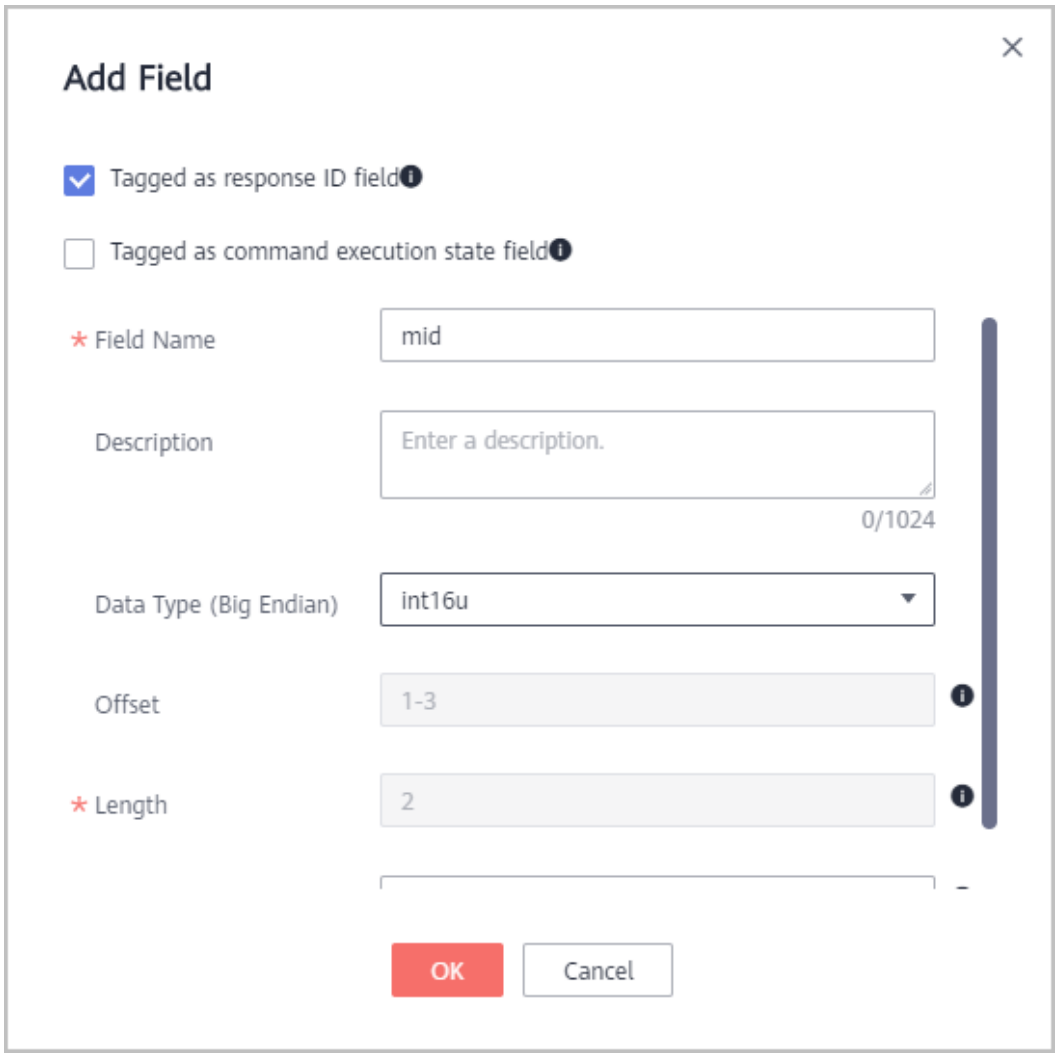
c. Add the **value** field to indicate the parameter value of the delivered command. For example, deliver the temperature threshold for a fire alarm. For details about the data type, length, default value, and offset, see 2.



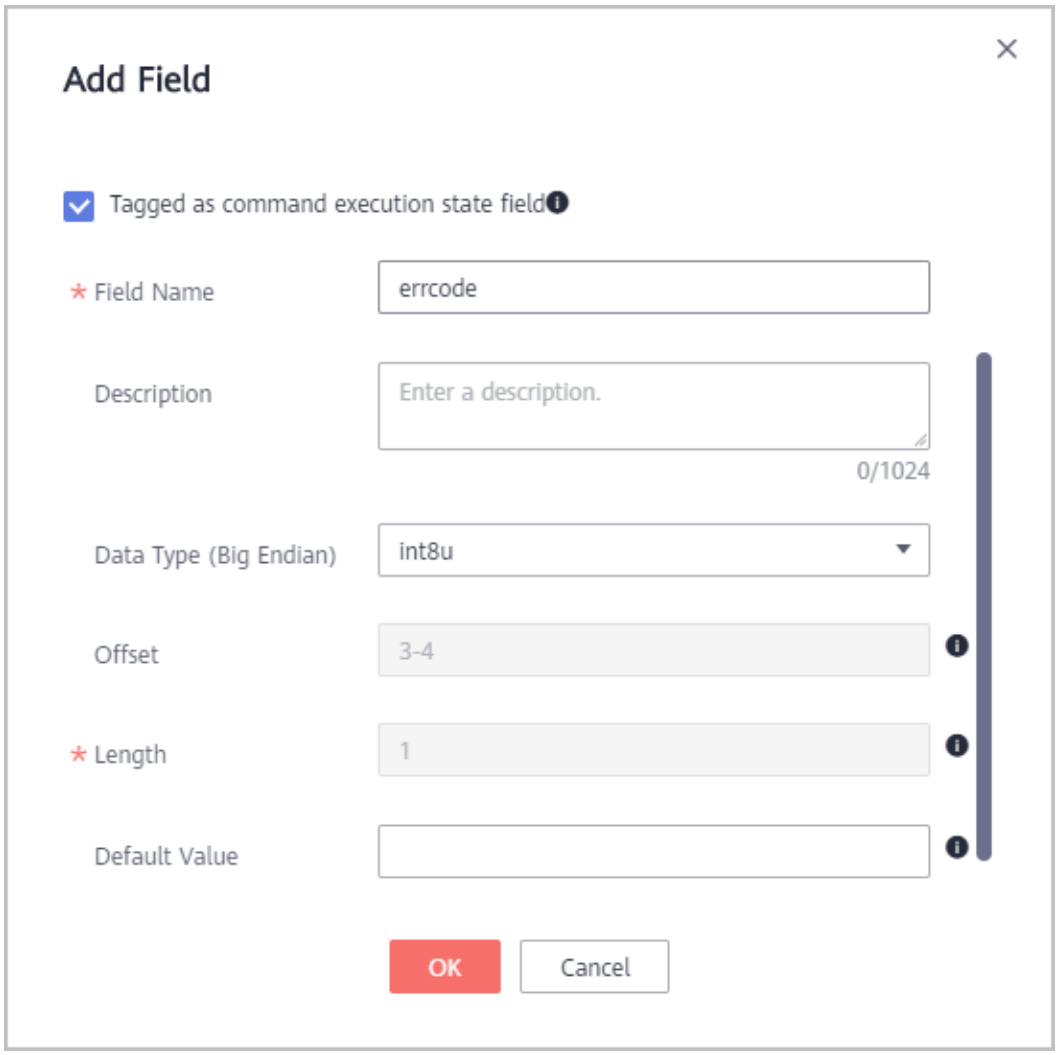
d. Click **Add Response Field** to add the **messageId** field, which indicates the message type. The command delivery response is an upstream message, which is differentiated from the data reporting message by the **messageId** field. The message type for reporting the temperature threshold of the fire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



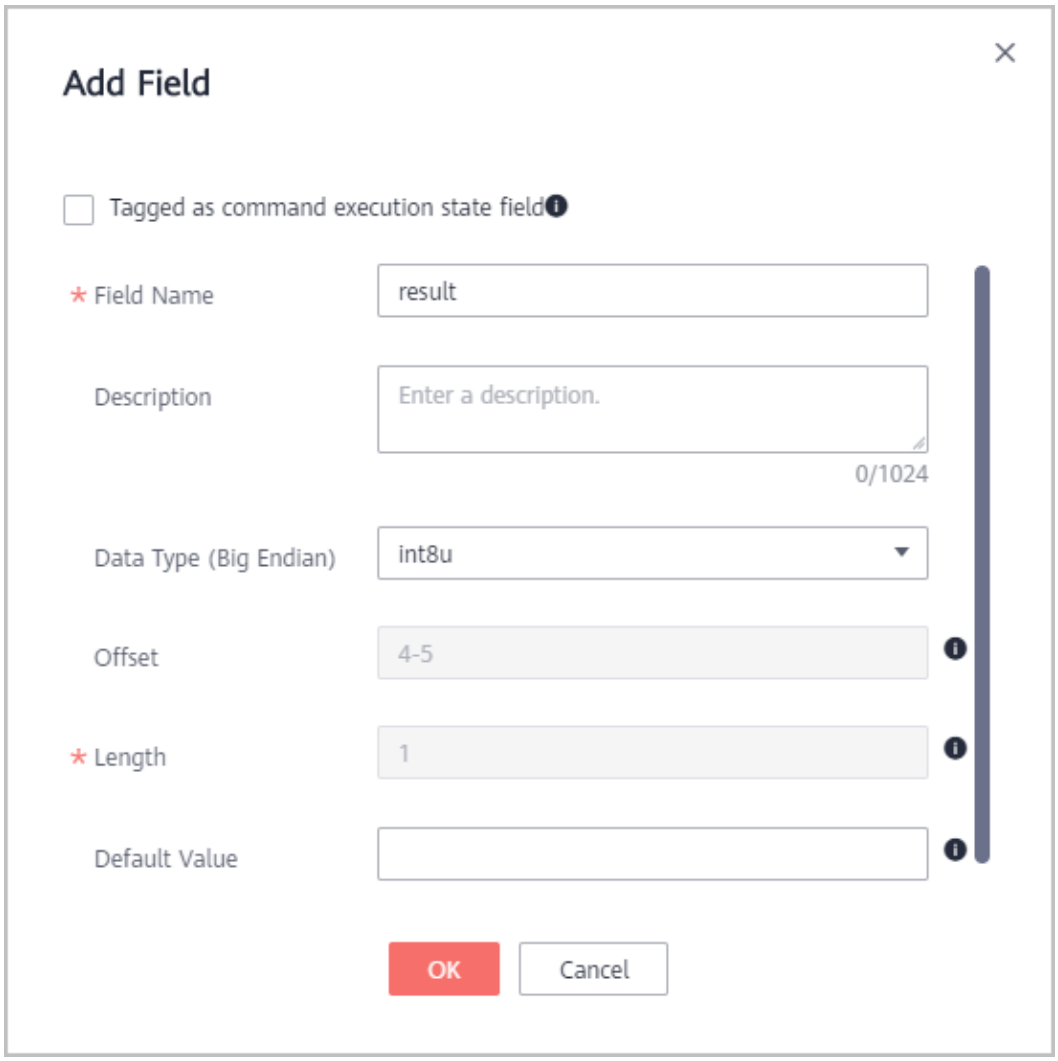
e. Add the **mid** field. This field must be the same as that in the command delivered by the IoT platform. It is used to associate the delivered command with the command execution result. The data type of the **mid** field is **int16u** by default. For details about the length, default value, and offset,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



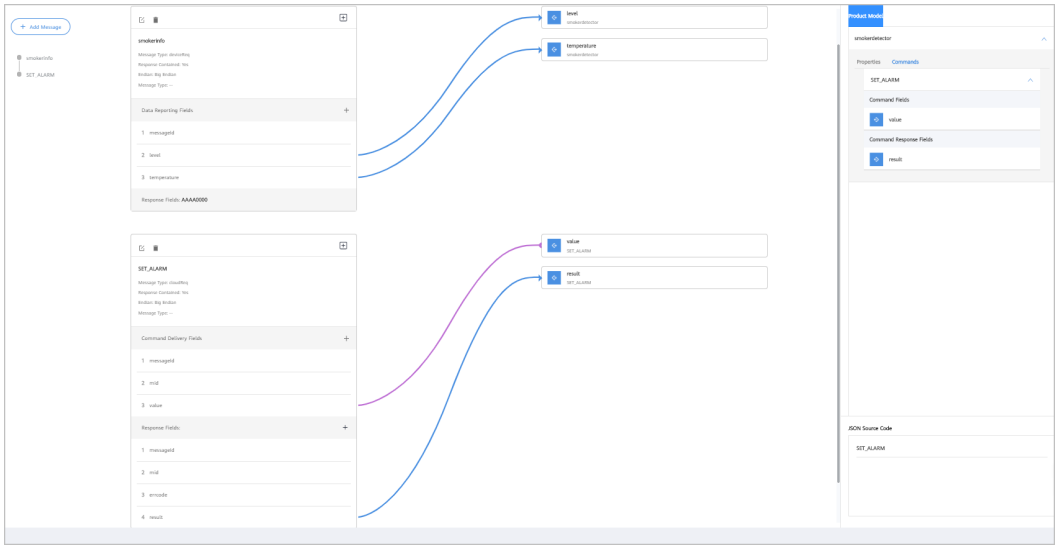
f. Add the **errcode** field to indicate the command execution status. **00** indicates success and **01** indicates failure. If this field is not carried, the command is executed successfully by default. The data type of the **errcode** field is **int8u** by default. For details about the length, default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



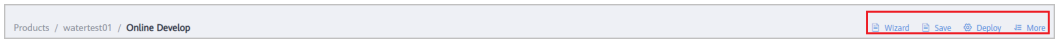
g. Add the **result** field to indicate the command execution result. For example, the device returns the current alarm threshold to the platform.



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



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



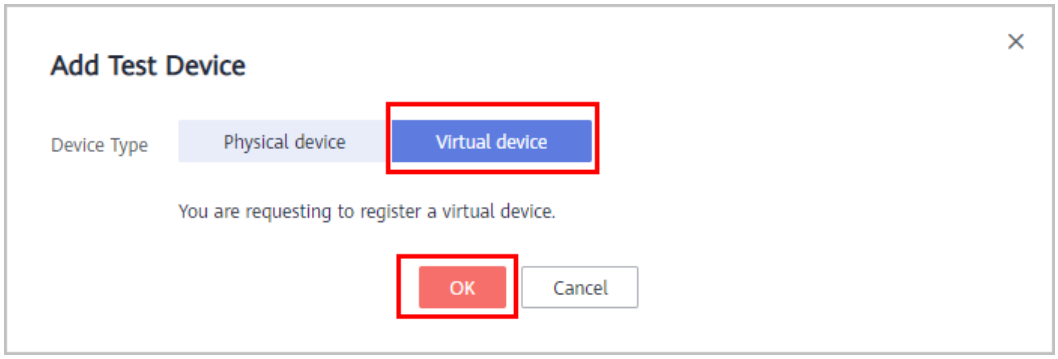
----End

Testing the Codec

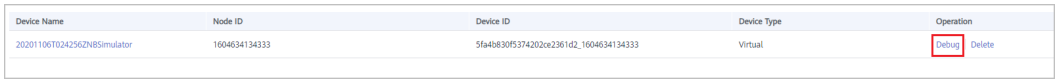
Step 1 On the product details page of the smoke detector, select **Online Debugging** and click **Add Test Device**.

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** and click **OK**. The virtual device name contains **Simulator**. Only one virtual device can be created for each product.



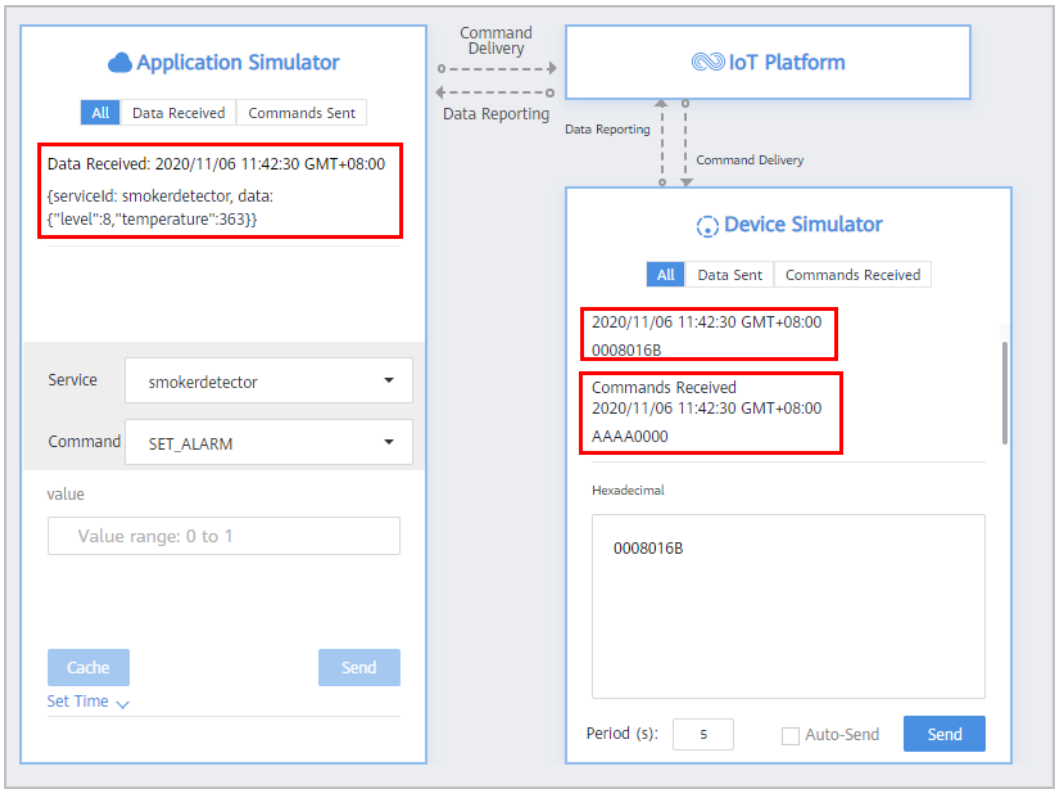
Step 3 Click **Debug** to access the debugging page.



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

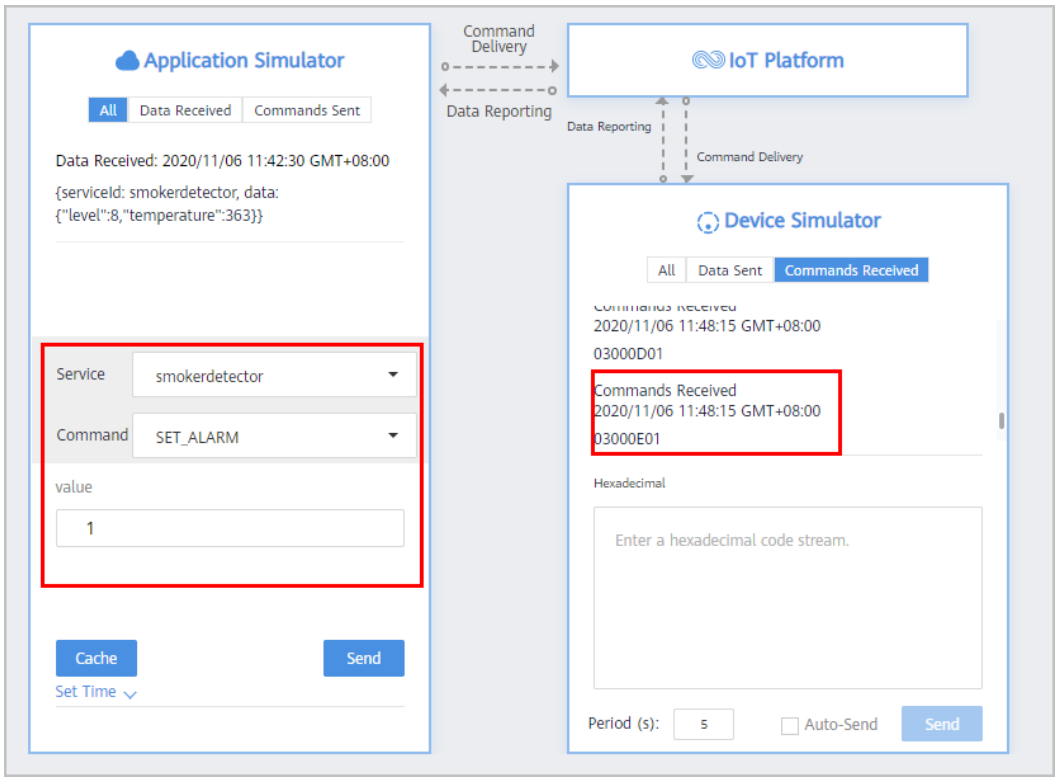
View the data reporting result ({level=8, temperature=363}) in **Application Simulator**. 8 is the decimal number converted from the hexadecimal number 08 and 363 from the hexadecimal number 018B.

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



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 **03000E01**. **03** indicate the **messageID** field, **000E** indicates the **mid** field, and **01** is the hexadecimal value converted from the decimal value **1**.



----End

Summary

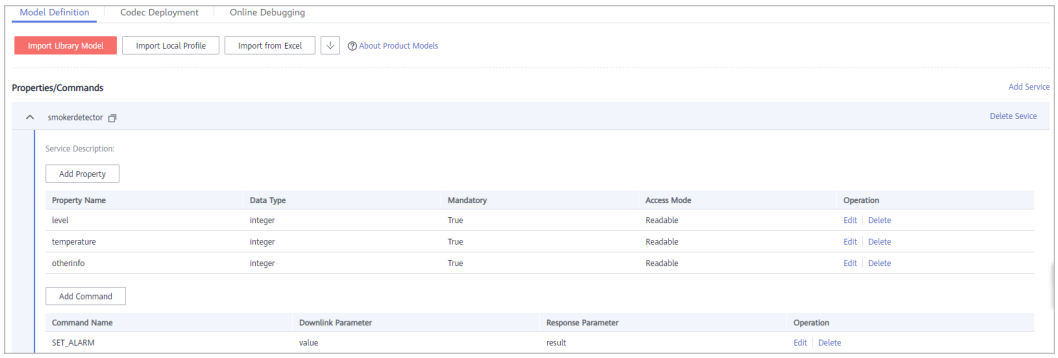
- If the codec needs to parse the command execution result, the **mid** field must be defined in the command and the command response.
- The length of the **mid** field in a command is two bytes. For each device, **mid** increases from 1 to 65535, and the corresponding code stream ranges from 0001 to 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:

Model Definition

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

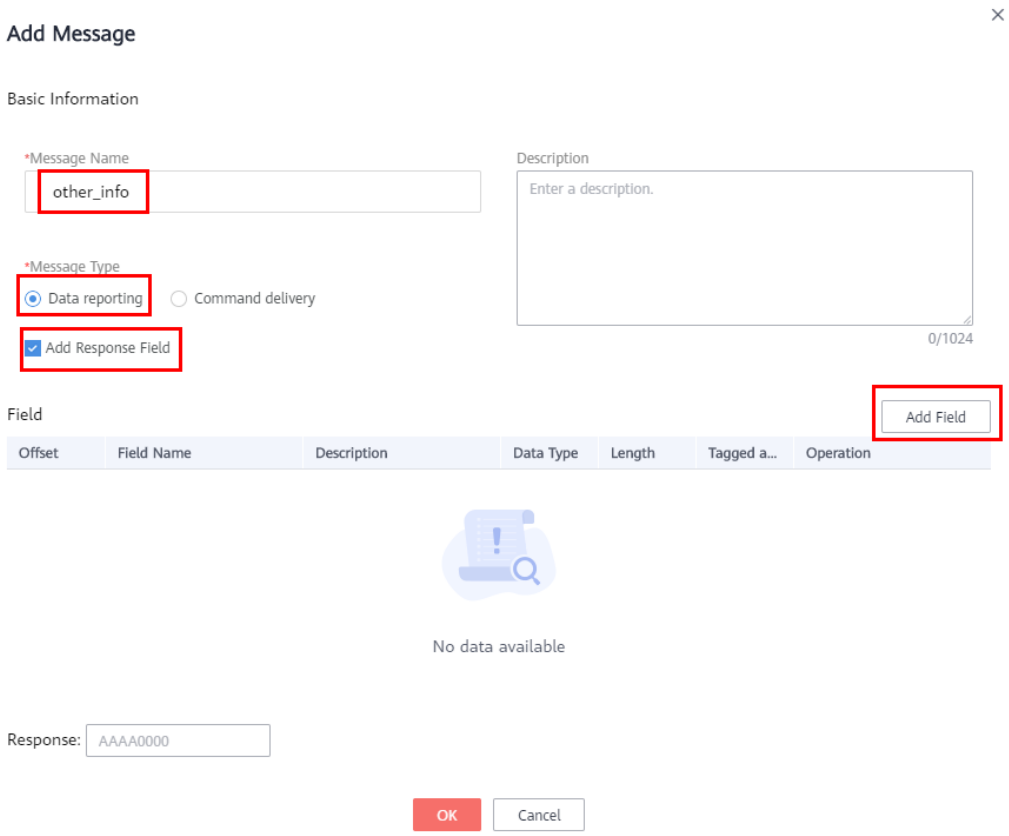


Developing a Codec

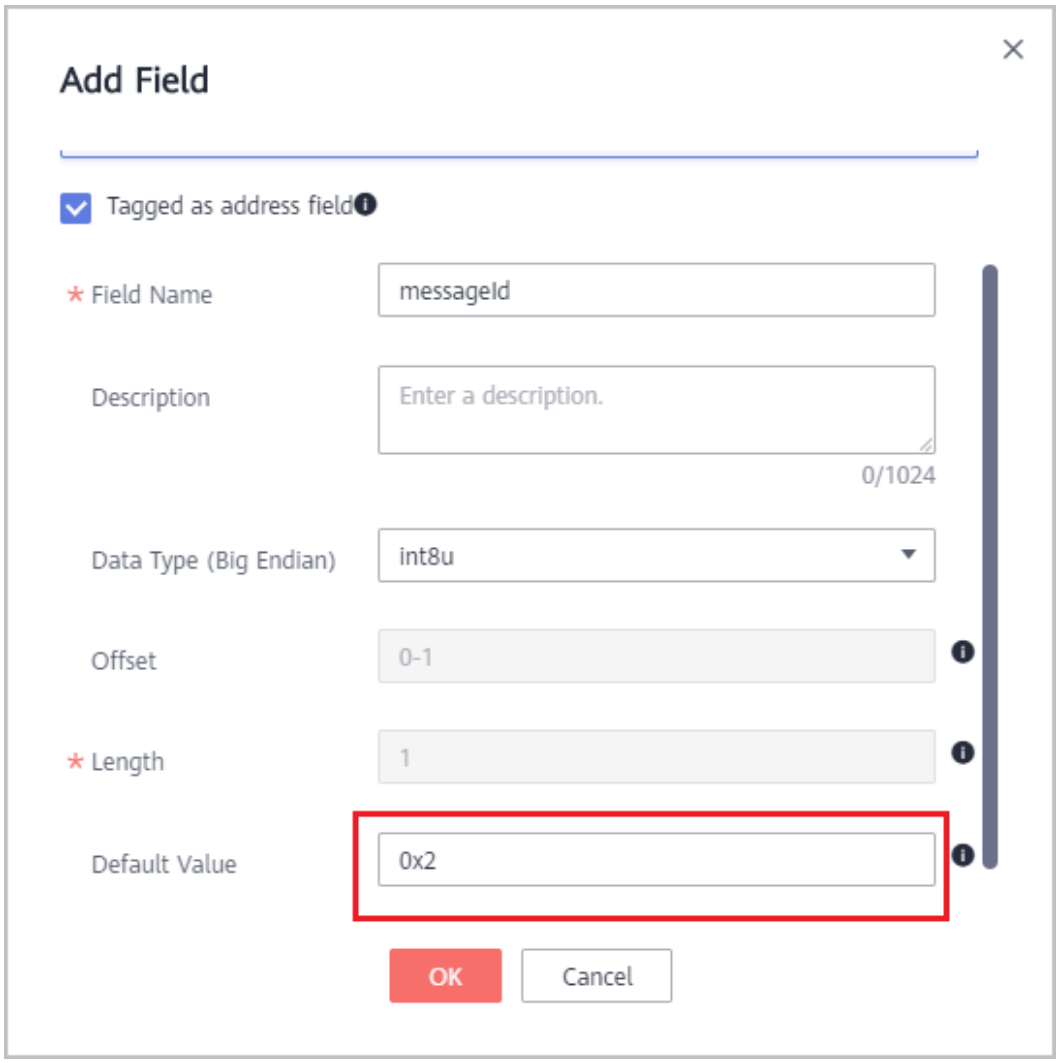
Step 1 On the product details page of the smoke detector, select **Codec Development** and click **Online Develop**.

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

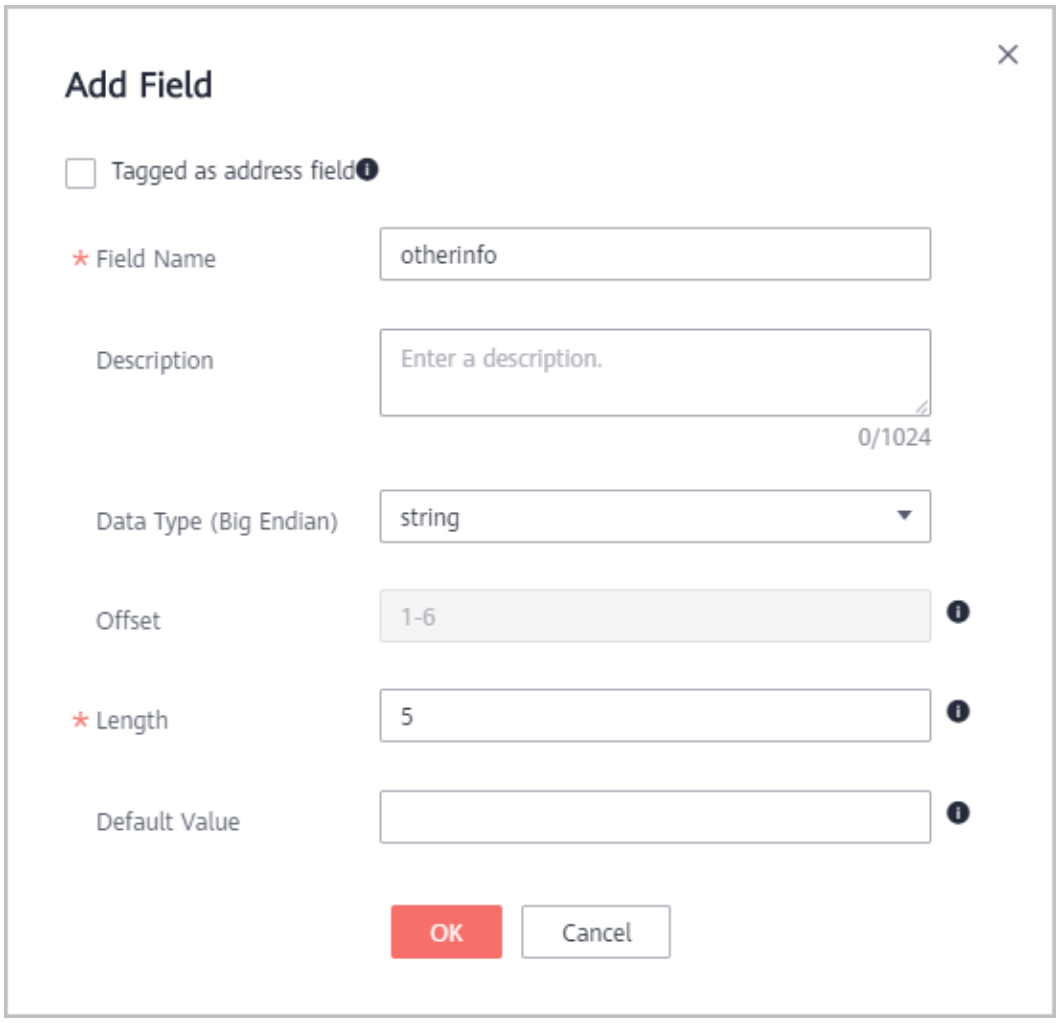
- **Message Name**: otherinfo
- **Message Type**: Data reporting
- **Add Response Field**: selected. After response fields are added, the platform delivers the response data set by the application to the device after receiving the data reported by the device.
- **Response**: AAAA0000 (default)



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 messageID, data type, length, default value, and offset, see **1**.

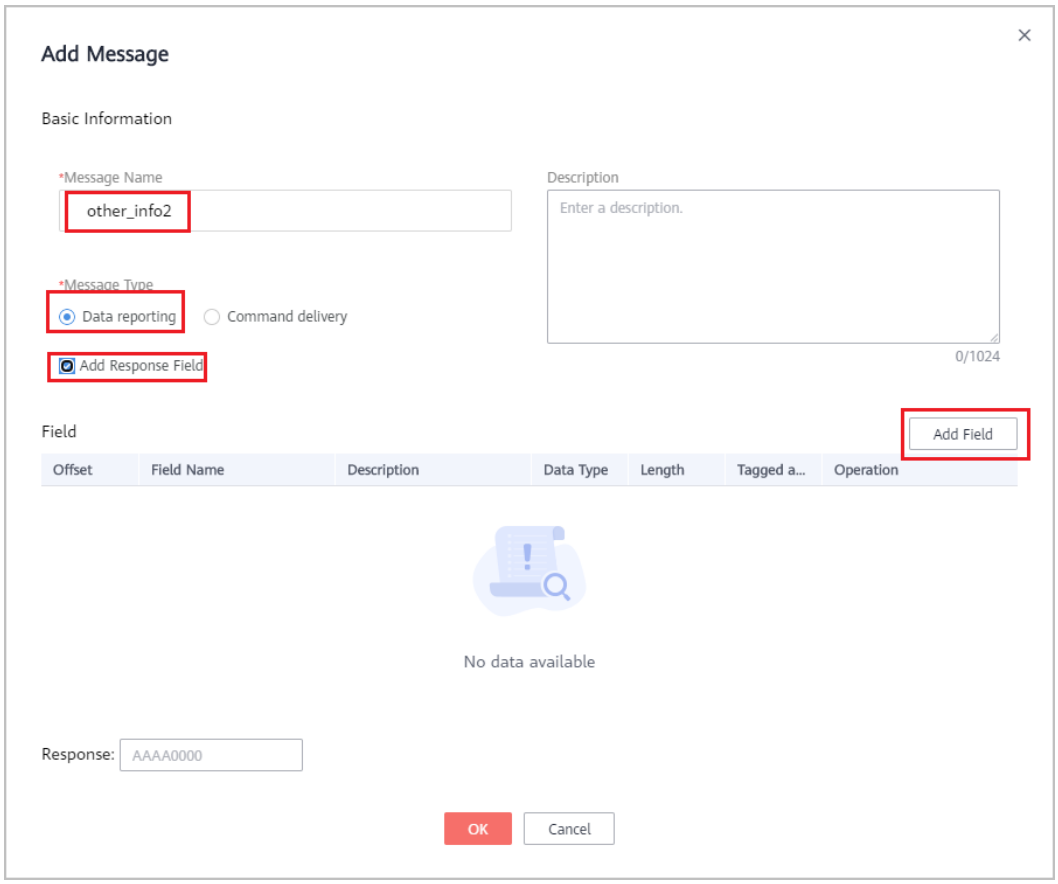


2. Add the **other_info** field to indicate the description of the string type. In this scenario, set **Data Type** to **string** and **Length** to **6**. For details about the field name, default value, and offset, see 2.

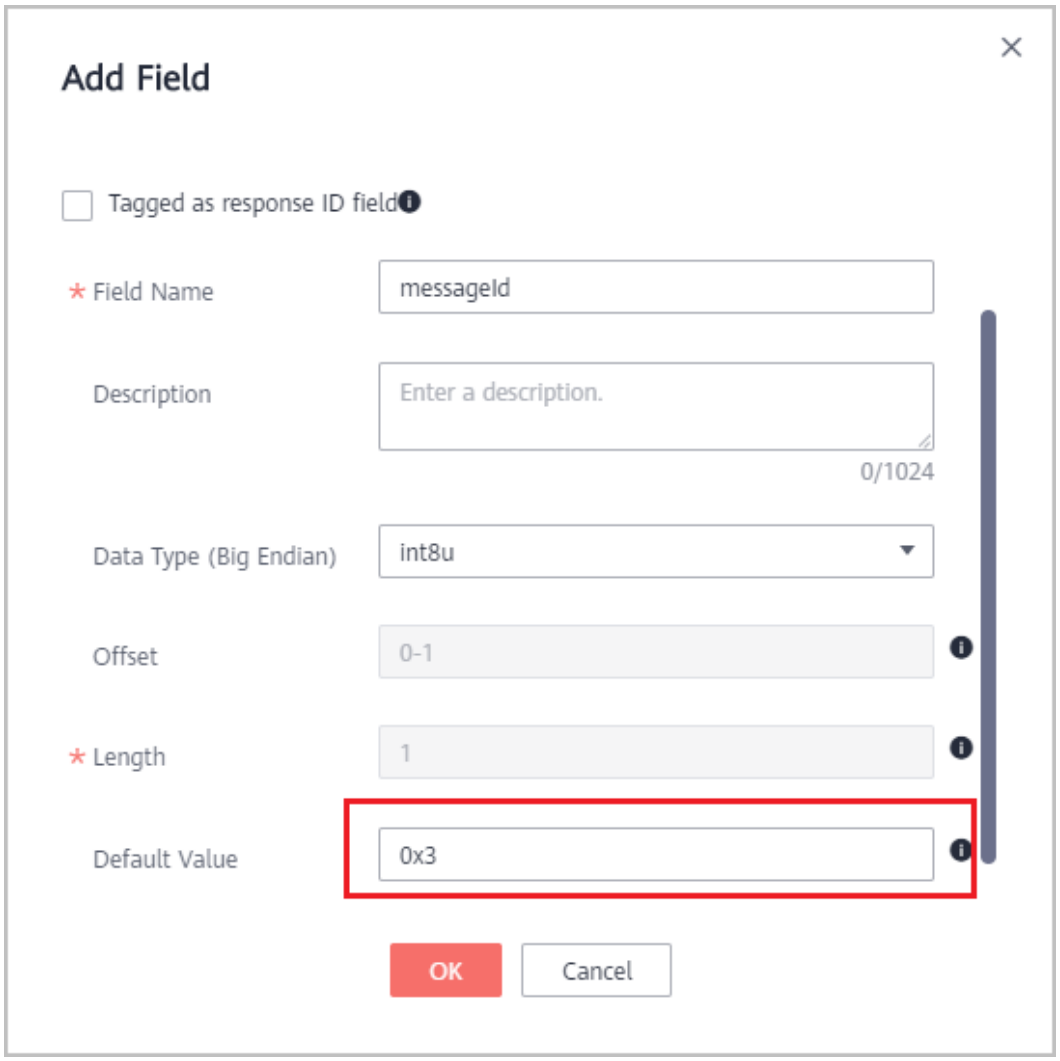


Step 3 Click **Add Message**, add the **other_info2** message name, and configure the data reporting message to report the description of the variable-length string type. This step is performed to decode the binary code stream message of variable-length strings uploaded by the device to the JSON format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

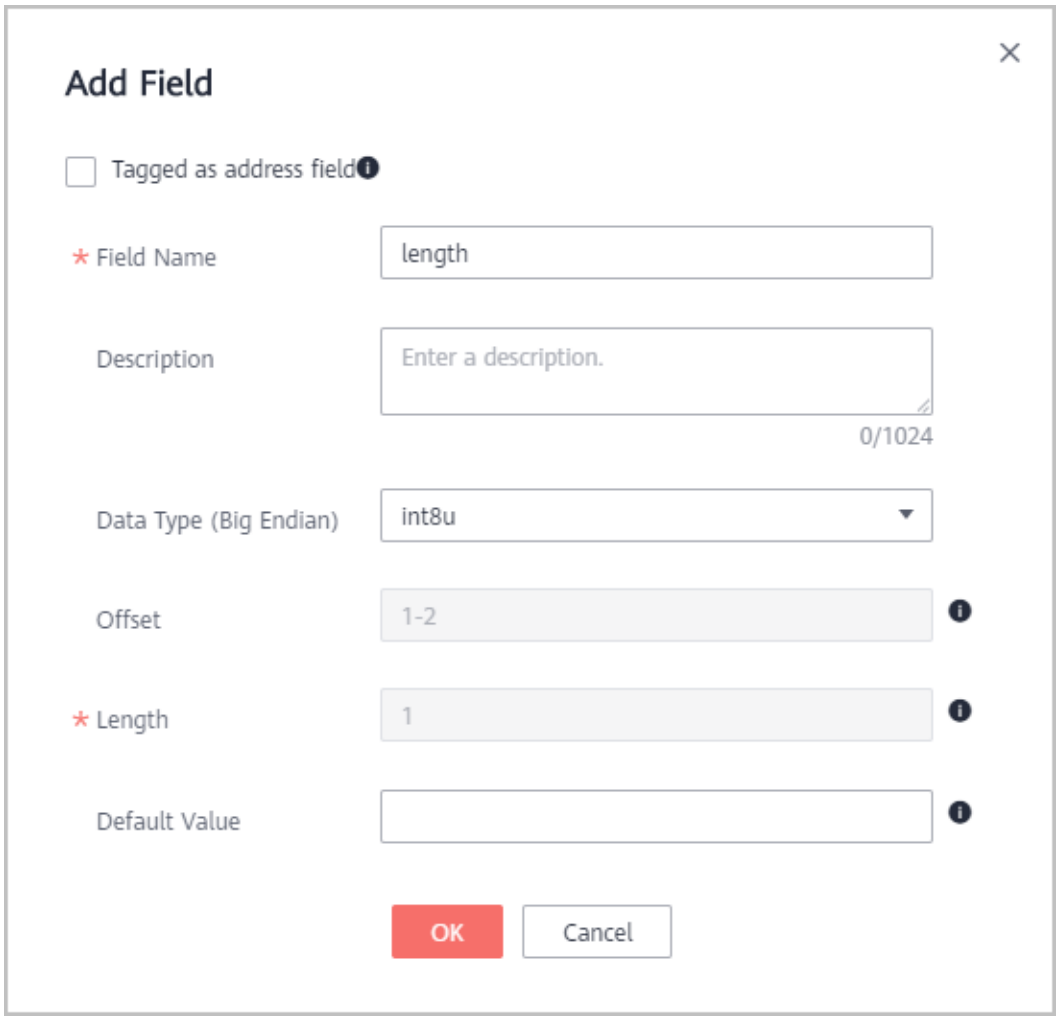
- **Message Name**: other_info2
- **Message Type**: Data reporting
- **Add Response Field**: selected. After response fields are added, the platform delivers the response data set by the application to the device after receiving the data reported by the device.
- **Response**: AAAA0000 (default)



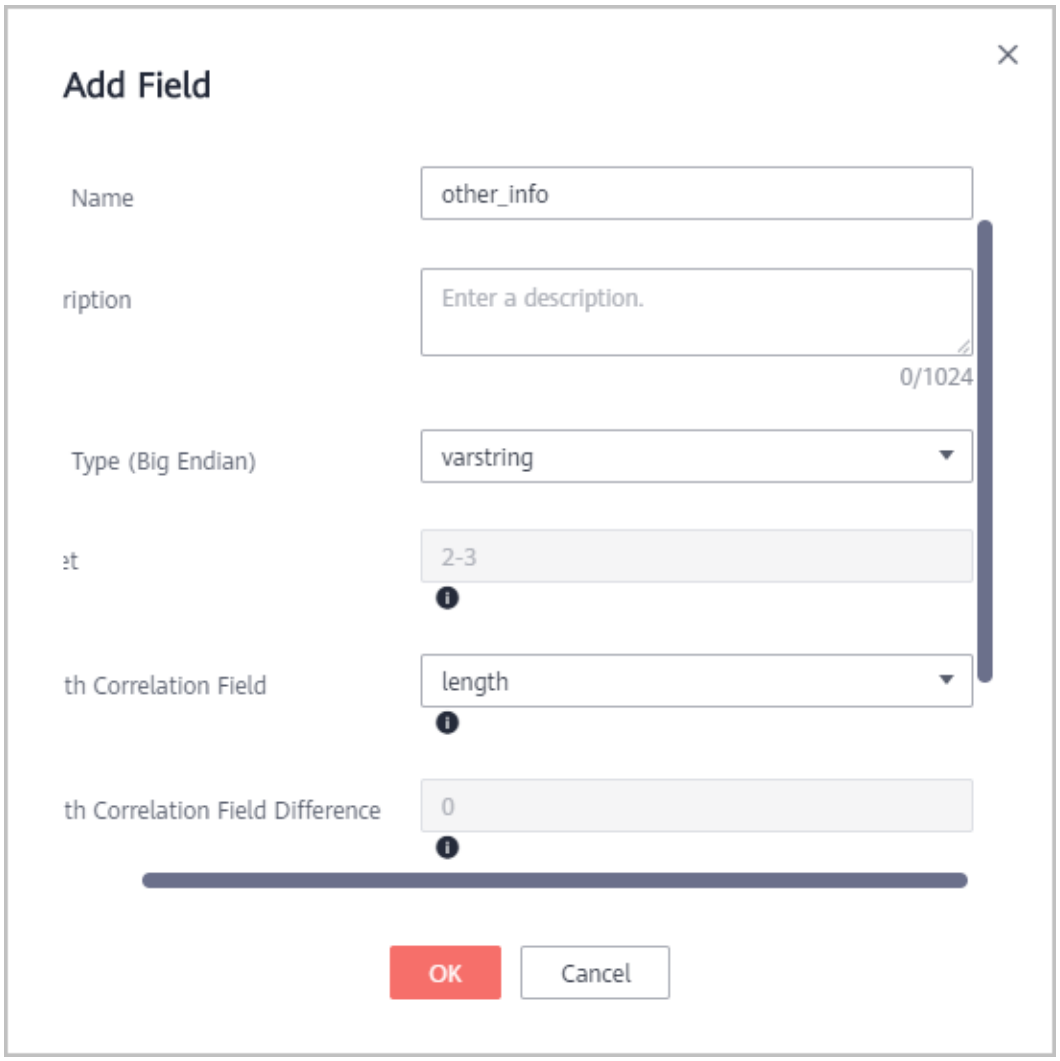
1. Add the **messageId** field to indicate the message type. In this scenario, the value **0x0** is used to identify the message that reports the fire severity and temperature, **0x1** is used to identify the message that reports only the temperature, and **0x3** is used to identify the message that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



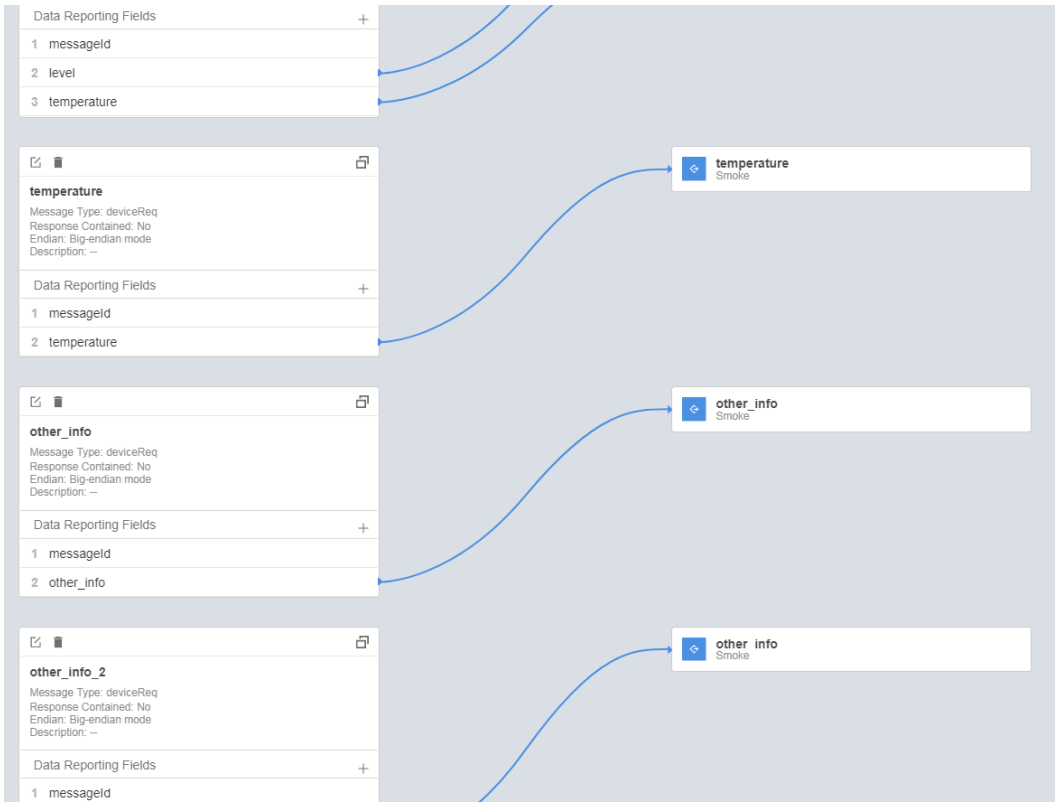
2. Add the **length** field to indicate the length of a variable-length string. **Data Type** is configured based on the length of the variable-length string. If the string contains 255 or fewer characters in this scenario, set this parameter to **int8u**. For details about the length, default value,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



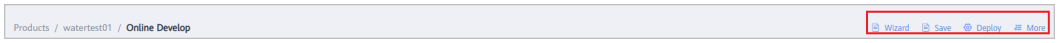
3. Add the **other_info** field and set **Data Type** to **varstring**, which indicates the description of the variable-length string type. Set **Length Correlation Field** to **length**. The values of **Length Correlation Field Difference** and **Length** are automatically filled. Retain the default value **0xff** for **Mask**. For details about the offset value, see **2**.



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



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



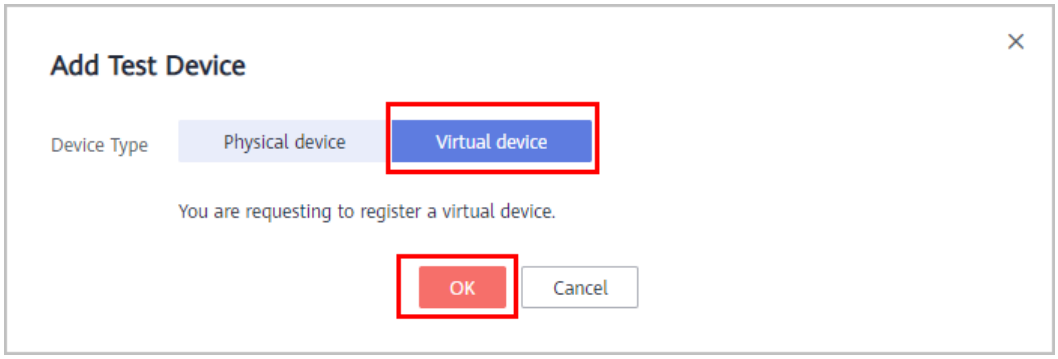
----End

Testing the Codec

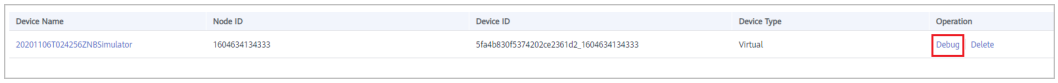
Step 1 On the product details page of the smoke detector, select **Online Debugging** and click **Add Test Device**.

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** and click **OK**. The virtual device name contains **Simulator**. Only one virtual device can be created for each product.



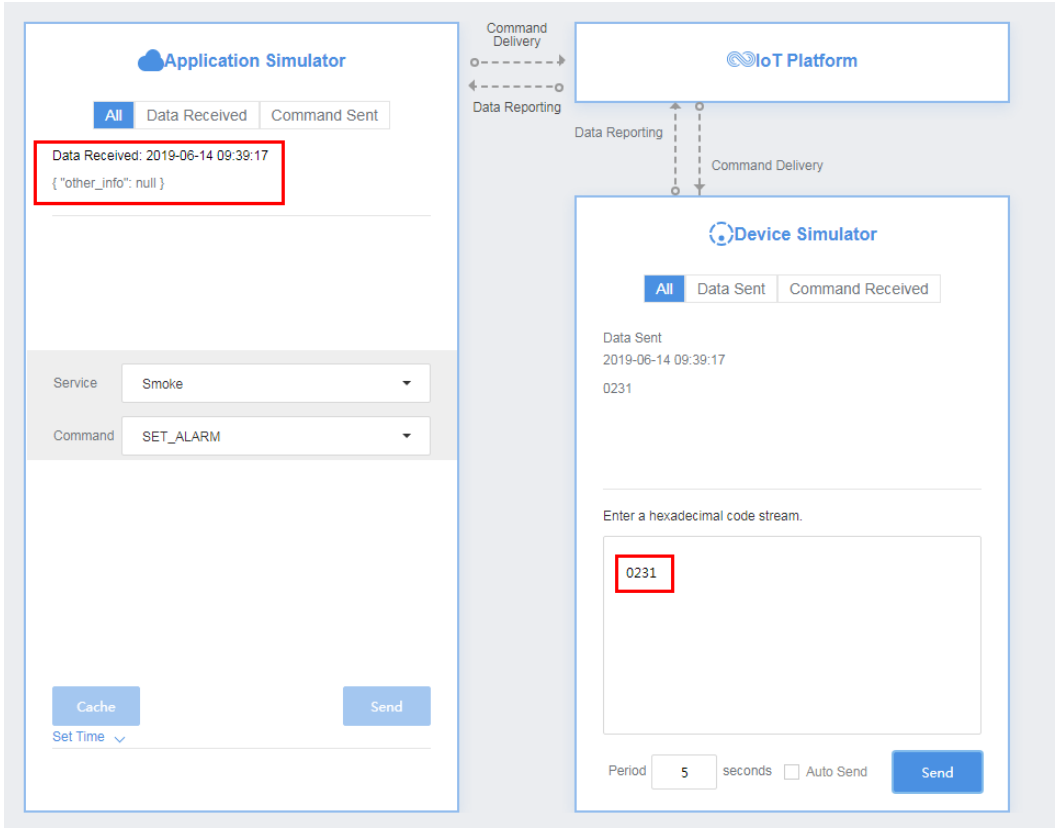
Step 3 Click **Debug** to access the debugging page.



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

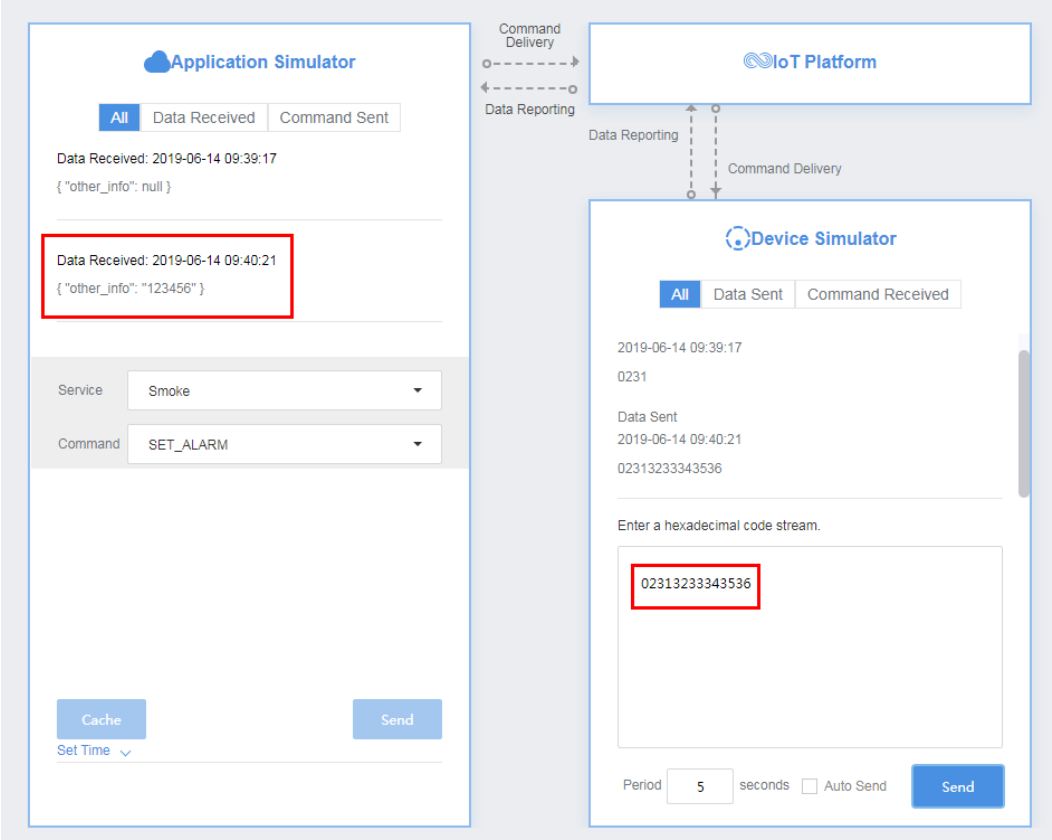
For example, a hexadecimal code stream (0231) is reported. **02** indicates the **messageId** field and specifies that this message reports the description of the string type. **31** indicates the description and its length is one byte.

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.



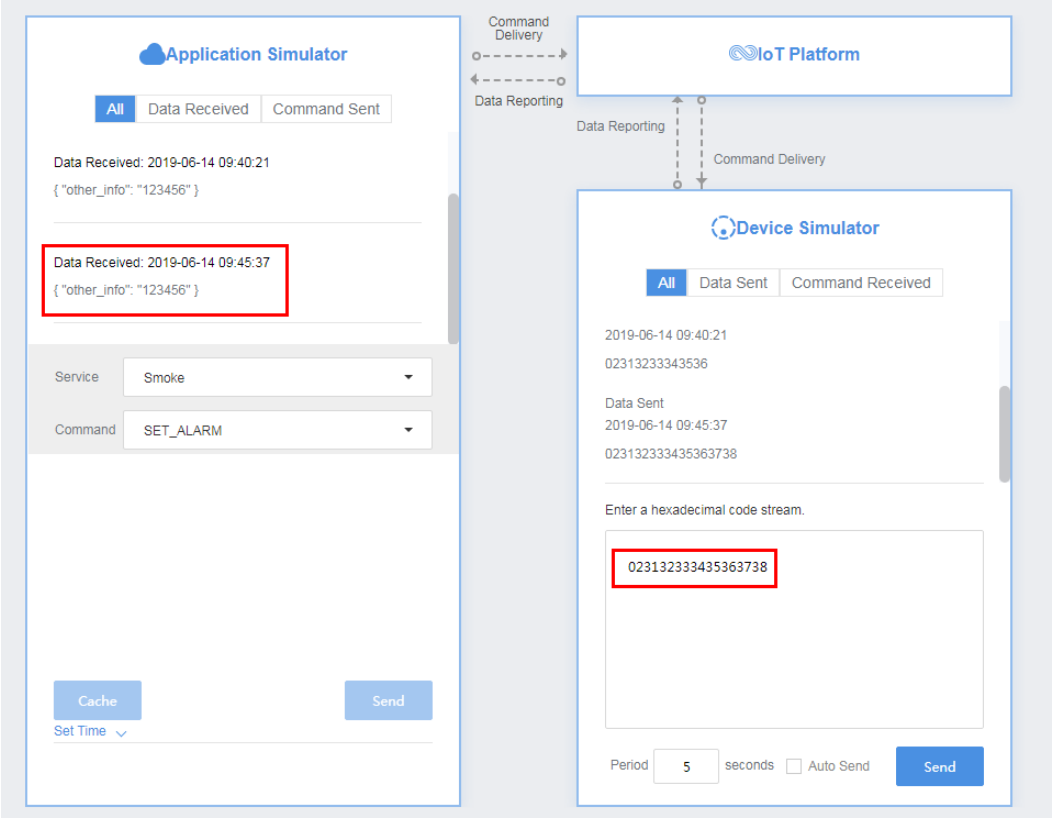
In the second hexadecimal code stream example (02313233343536), **02** indicates the **messageId** field and specifies that this message reports the description of the string type. **313233343536** indicates the description and its length is six bytes.

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.



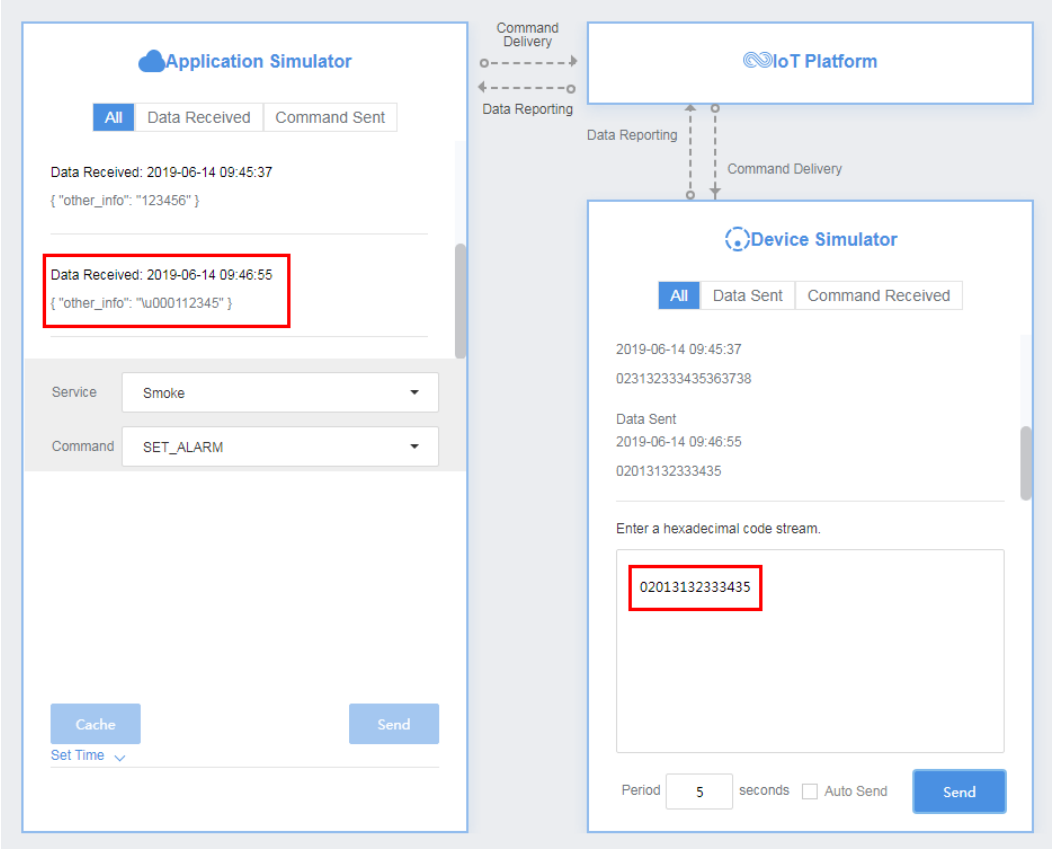
In the third hexadecimal code stream example (023132333435363738), **02** indicates the **messageId** field and specifies that this message reports the description of the string type. **3132333435363738** indicates the description and its length is eight bytes.

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.



In the fourth 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.

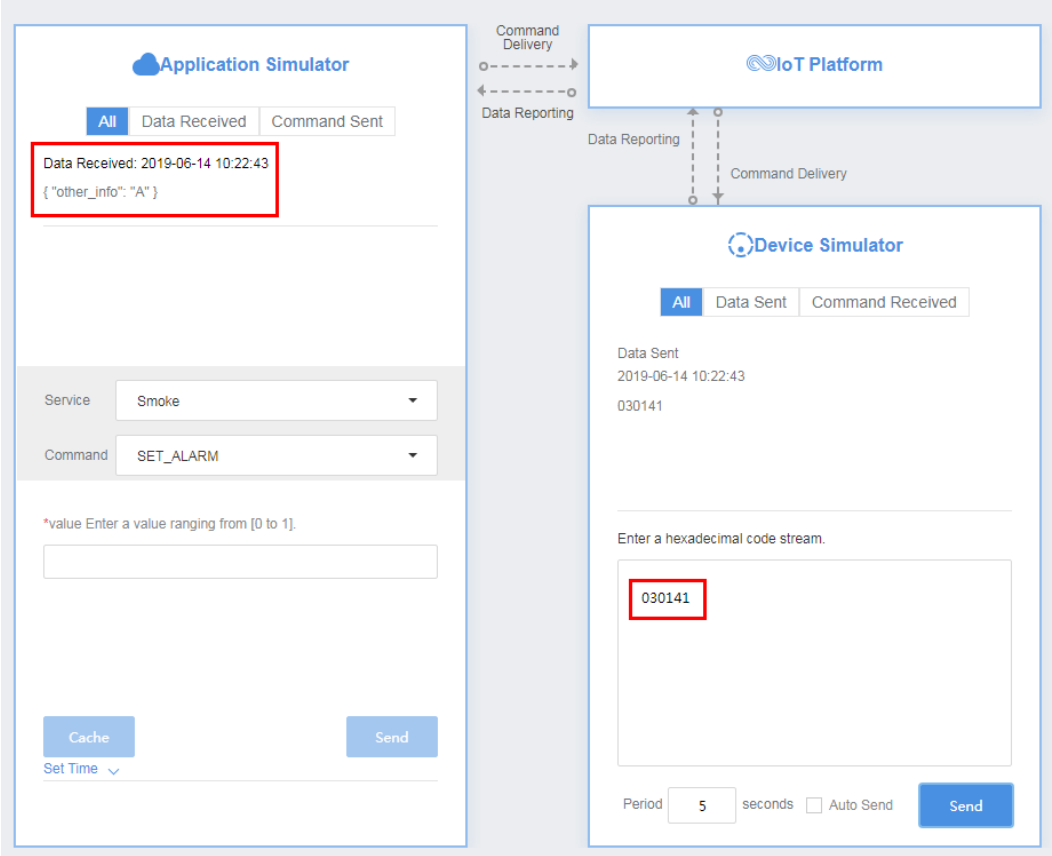
View the data reporting result ({other_info=\u000112345}) in **Application Simulator**. In the ASCII code table, **01** indicates **start of headline** which cannot be represented by specific characters. Therefore, 01 is parsed to \u0001.



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

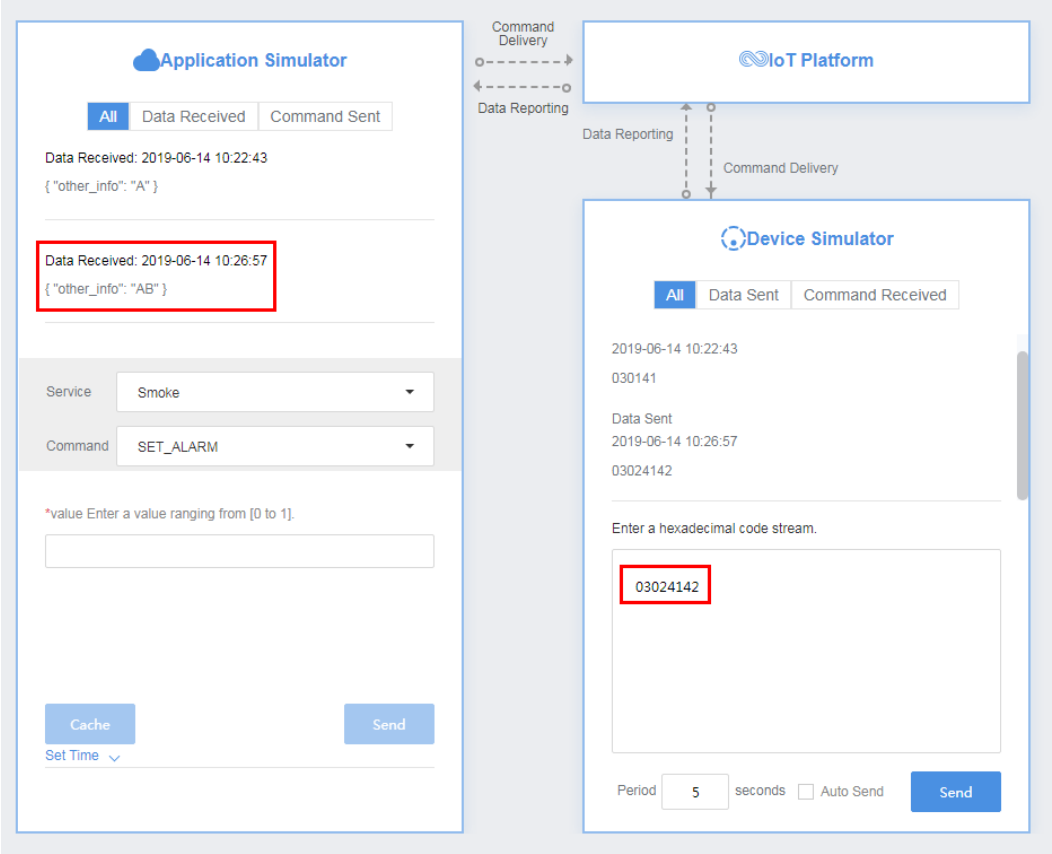
For example, a hexadecimal code stream (030141) is reported. In this code stream, **03** indicates the **messageId** field and specifies that this message reports the description of the variable-length string type. **01** indicates the length of the description (one byte) and its length is one byte.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



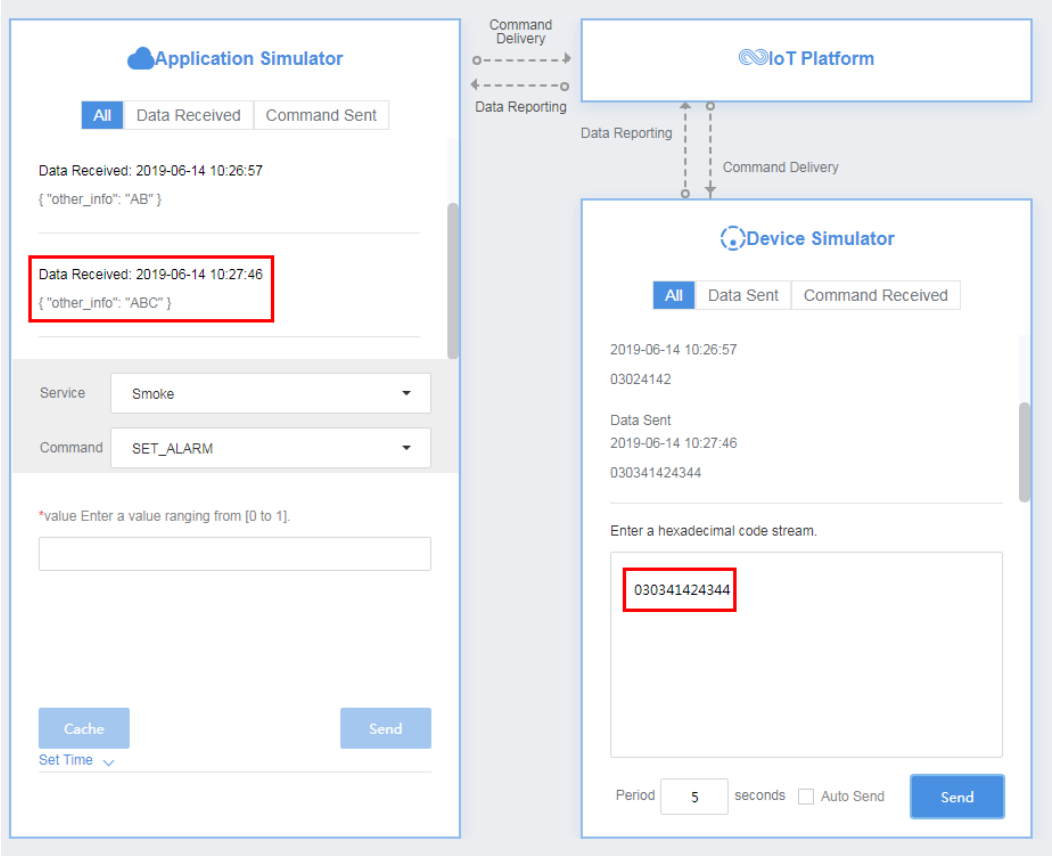
In the second hexadecimal code stream example (03024142), **03** indicates the **messageId** field and specifies that this message reports the description of the variable-length string type. **02** indicates the length of the description (two bytes) and its length is one byte. **4142** indicates the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



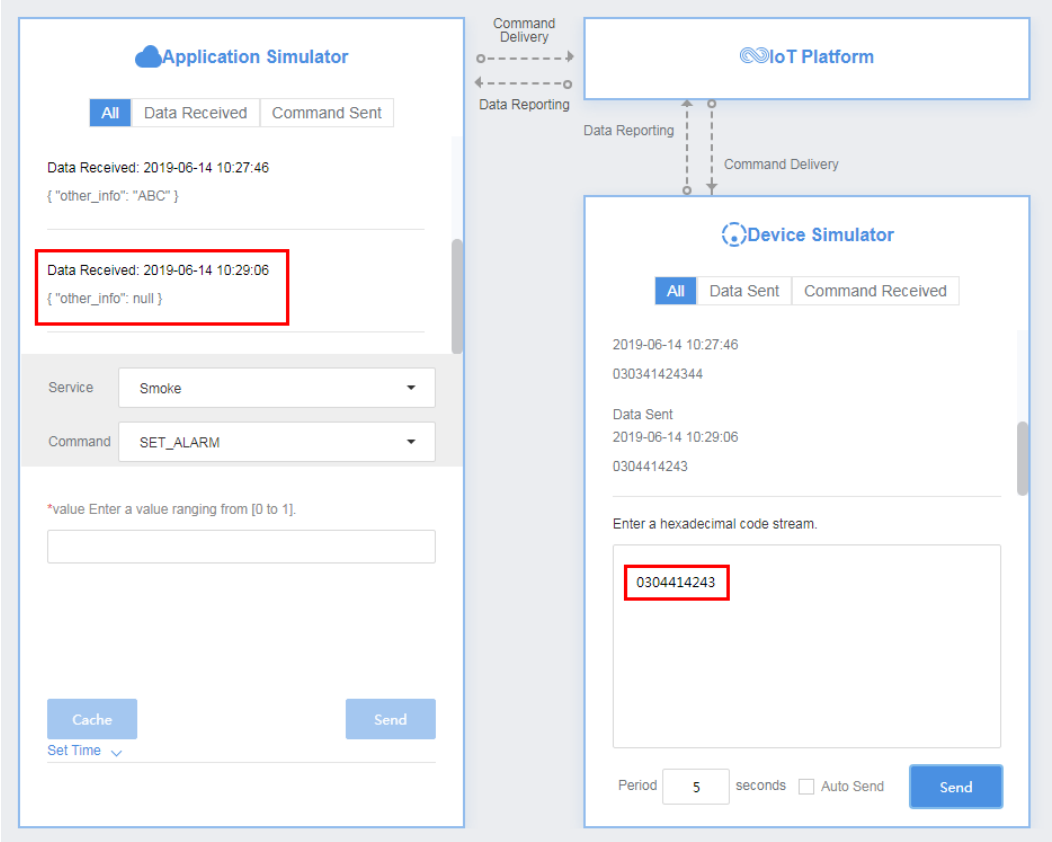
In the third hexadecimal code stream example (030341424344), **03** indicates the **messageId** field and specifies that this message reports the description of the variable-length string type. The second **03** indicates the length of the description (three bytes) and its length is one byte. **41424344**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



In the fourth 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.



----End

Summary

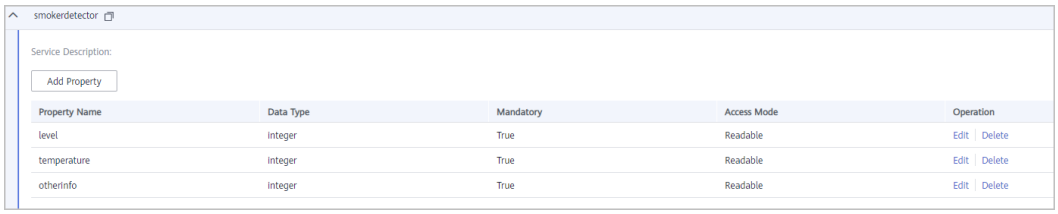
- When data is a string or a variable-length string, the codec processes the data based on the ASCII code. When data is reported, the hexadecimal code stream is decoded to a string. For example, 21 is parsed to an exclamation mark (!), 31 to 1, and 41 to A. When a command is delivered, the string is 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 in graphical mode 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:

Model Definition

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

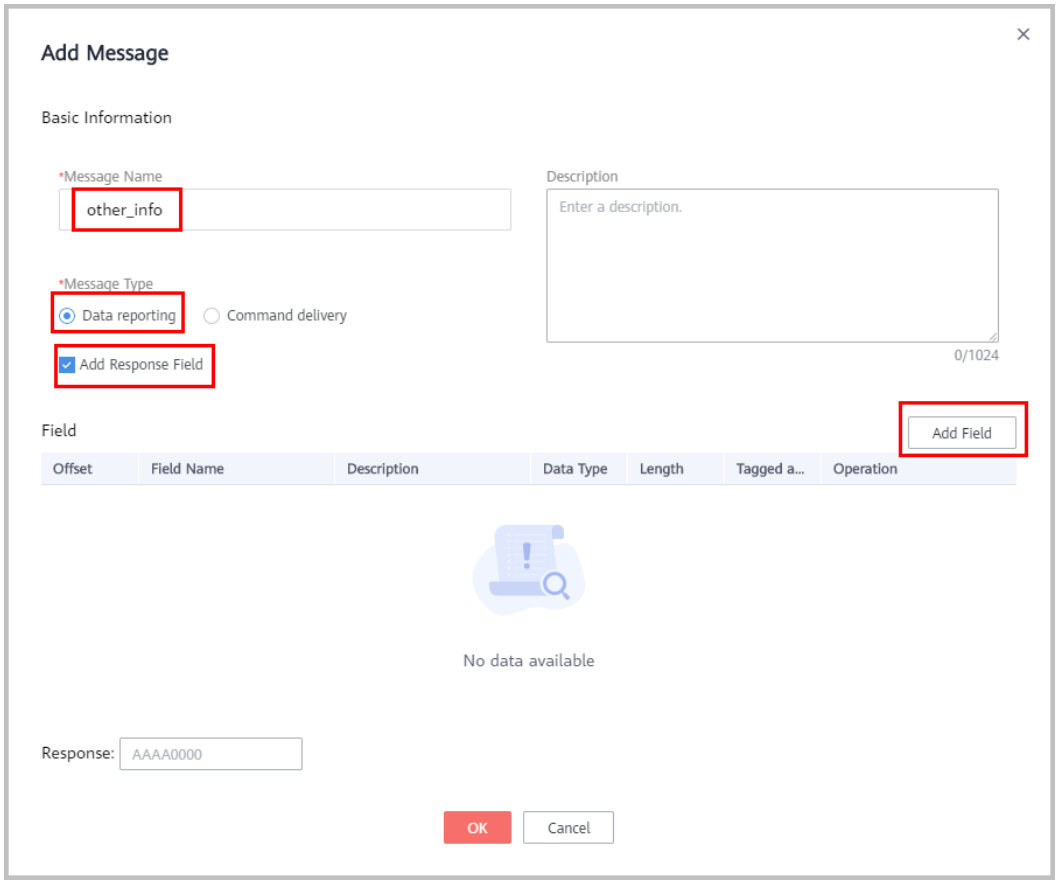


Developing a Codec

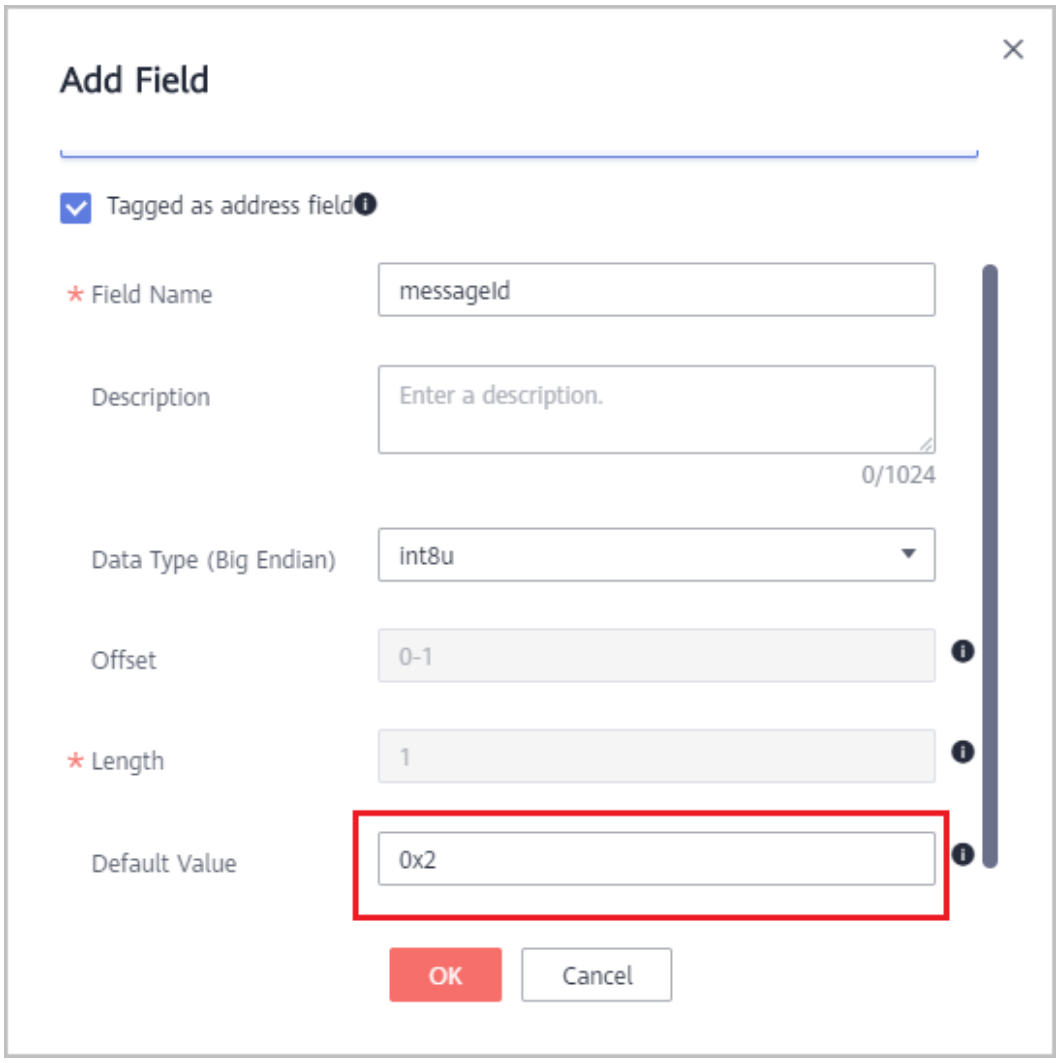
Step 1 On the product details page of the smoke detector, select **Codec Development** and click **Online Develop**.

Step 2 Click **Add Message** to add the **otherinfo** 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:

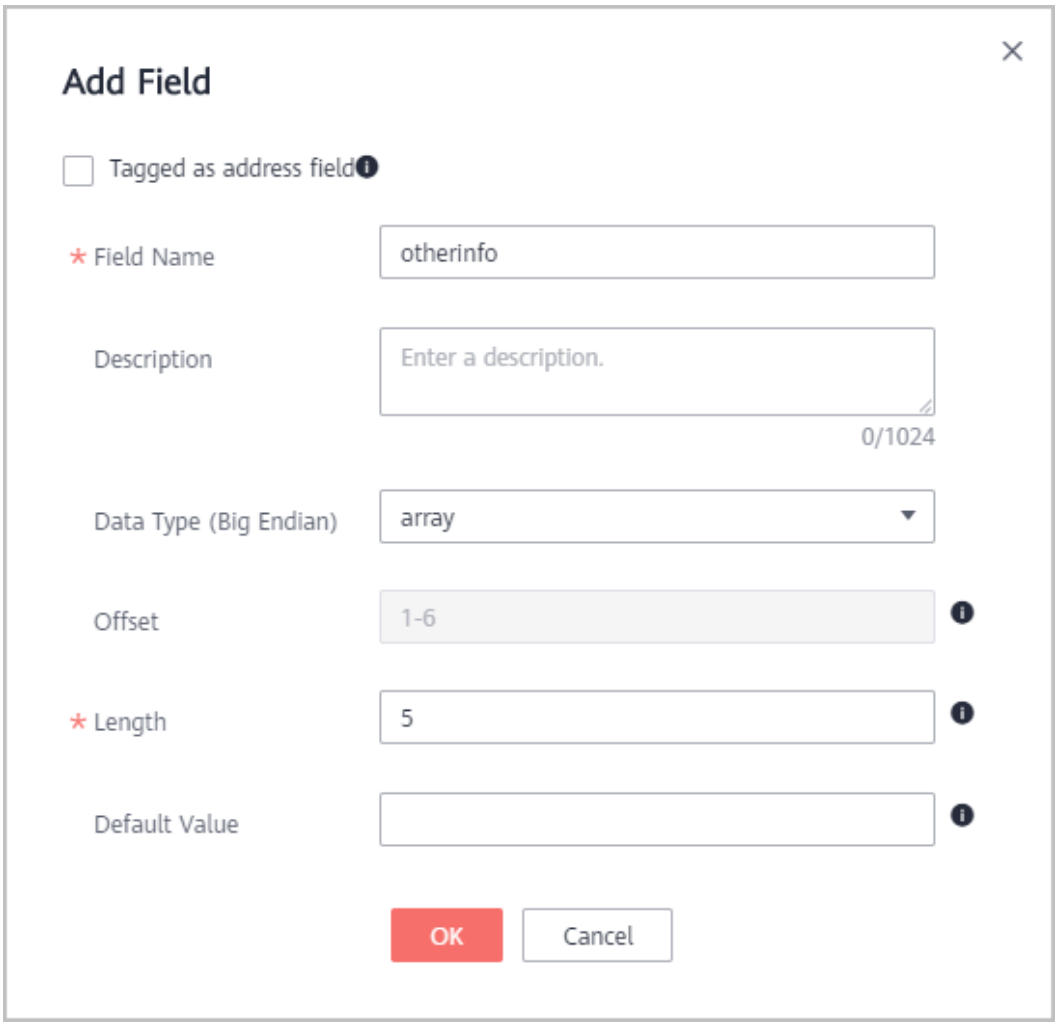
- Message Name: otherinfo
- Message Type: Data reporting
- Add Response Field: selected. After response fields are added, the platform delivers the response data set by the application to the device after receiving the data reported by the device.
- Response: AAAA0000 (default)



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 messageID, data type, length, default value, and offset, see 1.

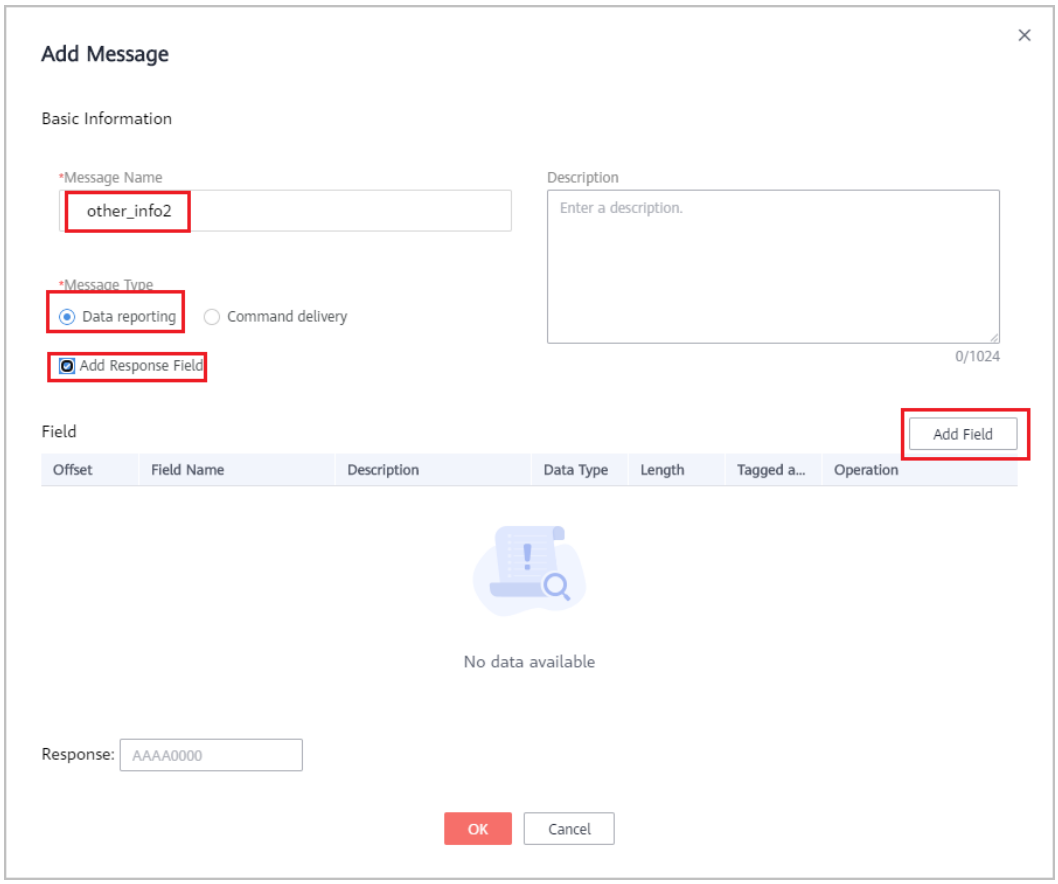


2. Add the **other_info** field and set **Data Type** to **array**, which indicates the description of the array type. In this scenario, set **Length** to **5**. For details about the field name, default value, and offset, see 2.

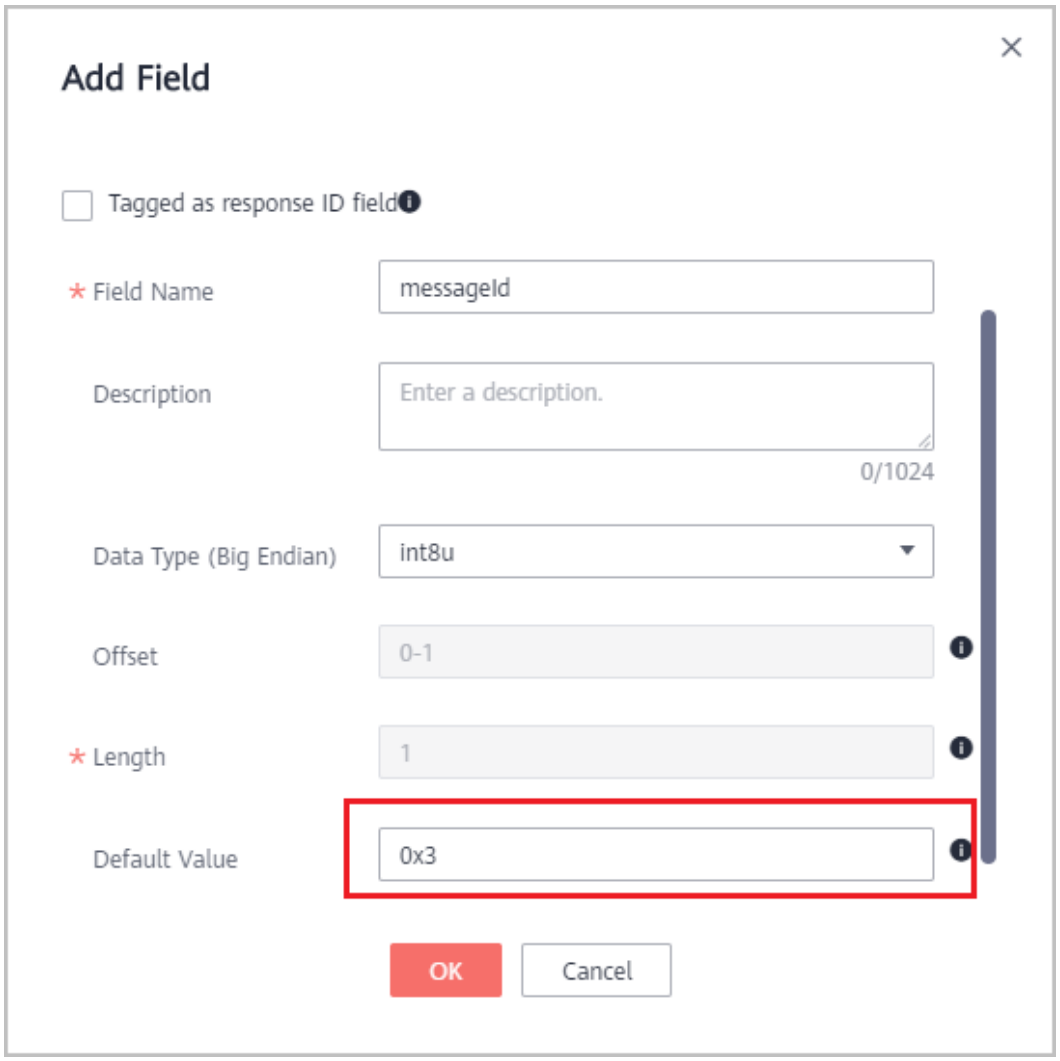


Step 3 Click **Add Message** to add the **other_info2** message and report the description of the variable-length array type. This step is performed to decode the binary code stream message of variable-length arrays uploaded by the device to the JSON format so that the platform can understand the message.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

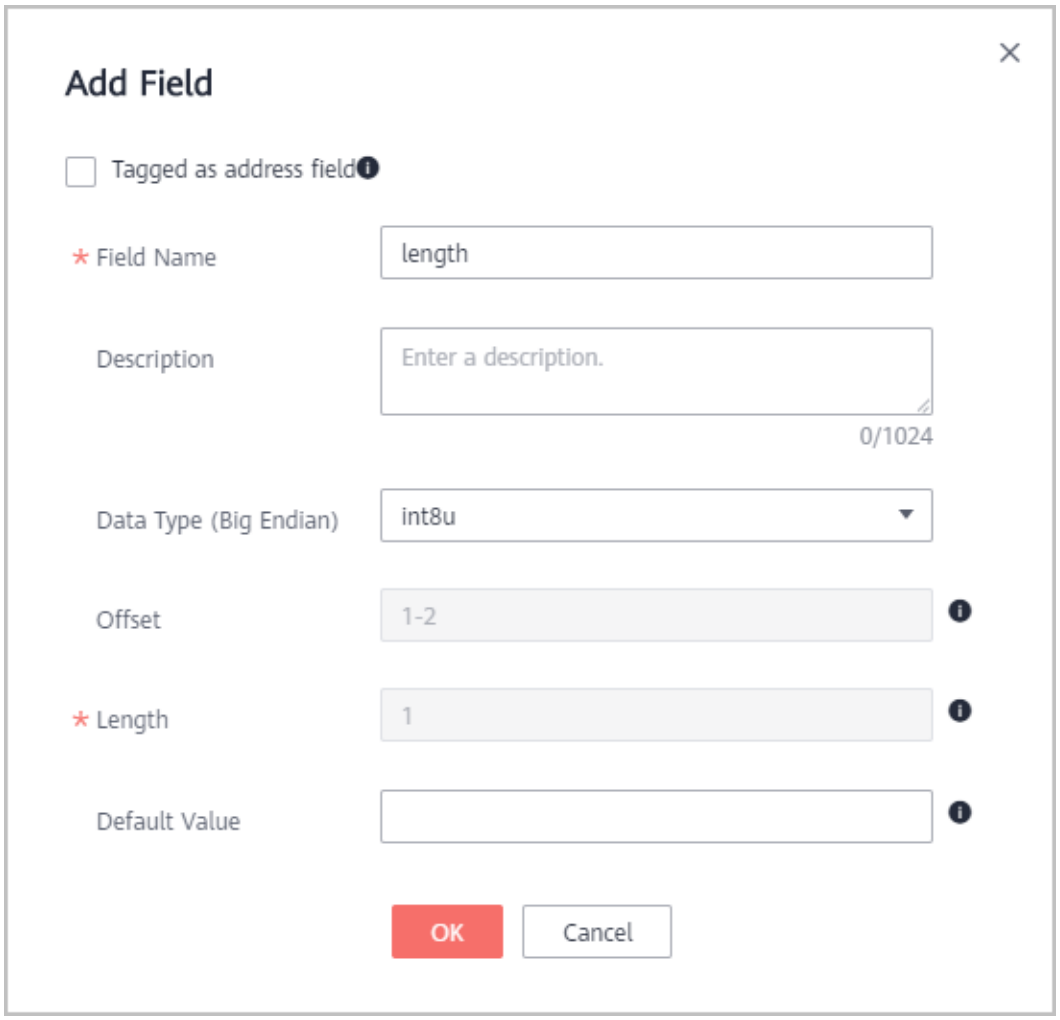
- **Message Name**: other_info2
- **Message Type**: Data reporting
- **Add Response Field**: selected. After response fields are added, the platform delivers the response data set by the application to the device after receiving the data reported by the device.
- **Response**: AAAA0000 (default)



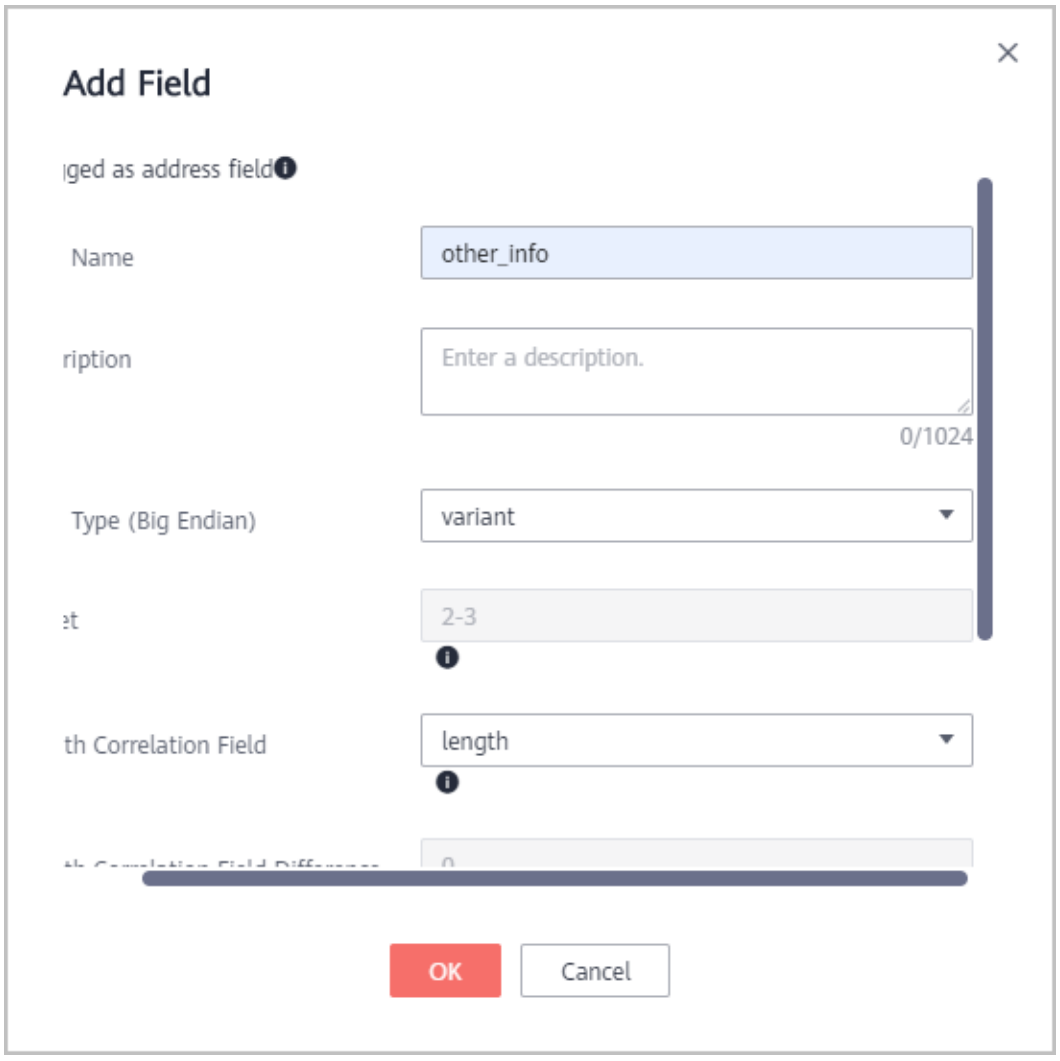
1. Click **Add Field** to add the **messageId** field, which indicates the message type. In this scenario, the value **0x0** is used to identify the message that reports the fire severity and temperature, **0x1** is used to identify the message that reports only the temperature, and **0x3** is used to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



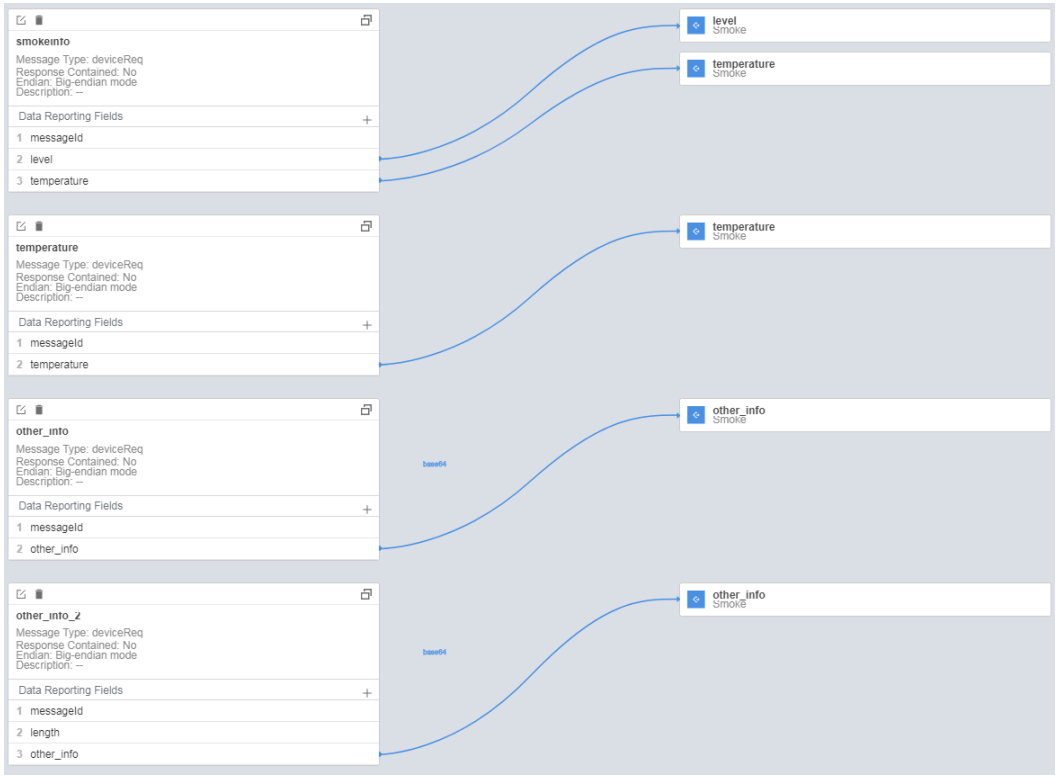
2. Add the **length** field to indicate the length of an array. **Data Type** is configured based on the length of the variable-length array. If the array contains 255 or fewer characters, set this parameter to **int8u**. For details about the length, default value, and offset, see 2.



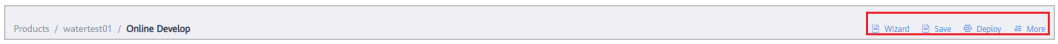
3. Add the **other_info** field and set **Data Type** to **variant**, which indicates the description of the variable-length array type. Set **Length Correlation Field** to **length**. The values of **Length Correlation Field Difference** and **Length** are automatically filled. Retain the default value **0xff** for **Mask**. For details about the offset value, see **2**.



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



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



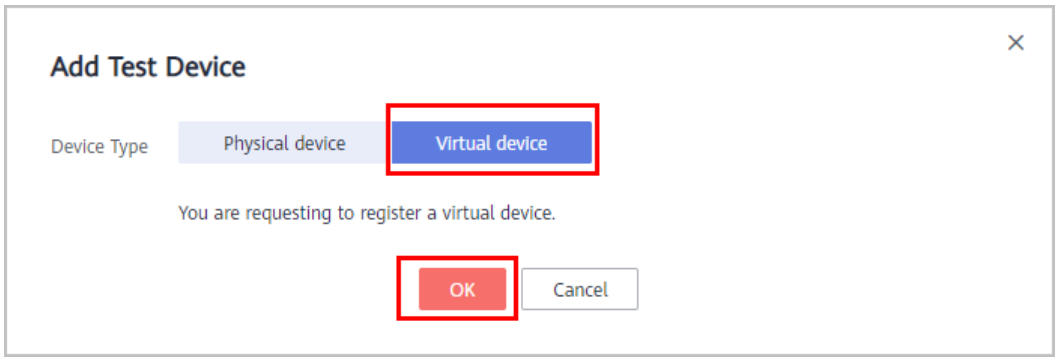
----End

Testing the Codec

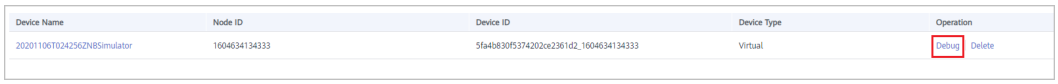
Step 1 On the product details page of the smoke detector, select **Online Debugging** and click **Add Test Device**.

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** and click **OK**. The virtual device name contains **Simulator**. Only one virtual device can be created for each product.



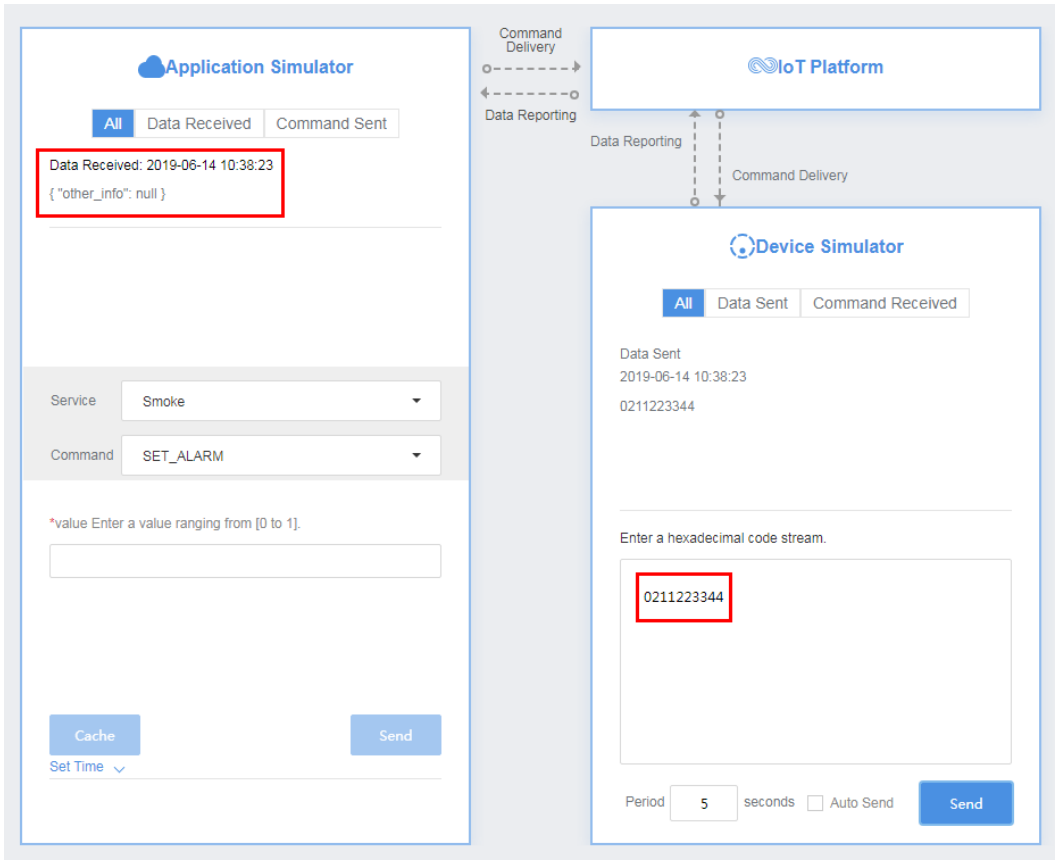
Step 3 Click **Debug** to access the debugging page.



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

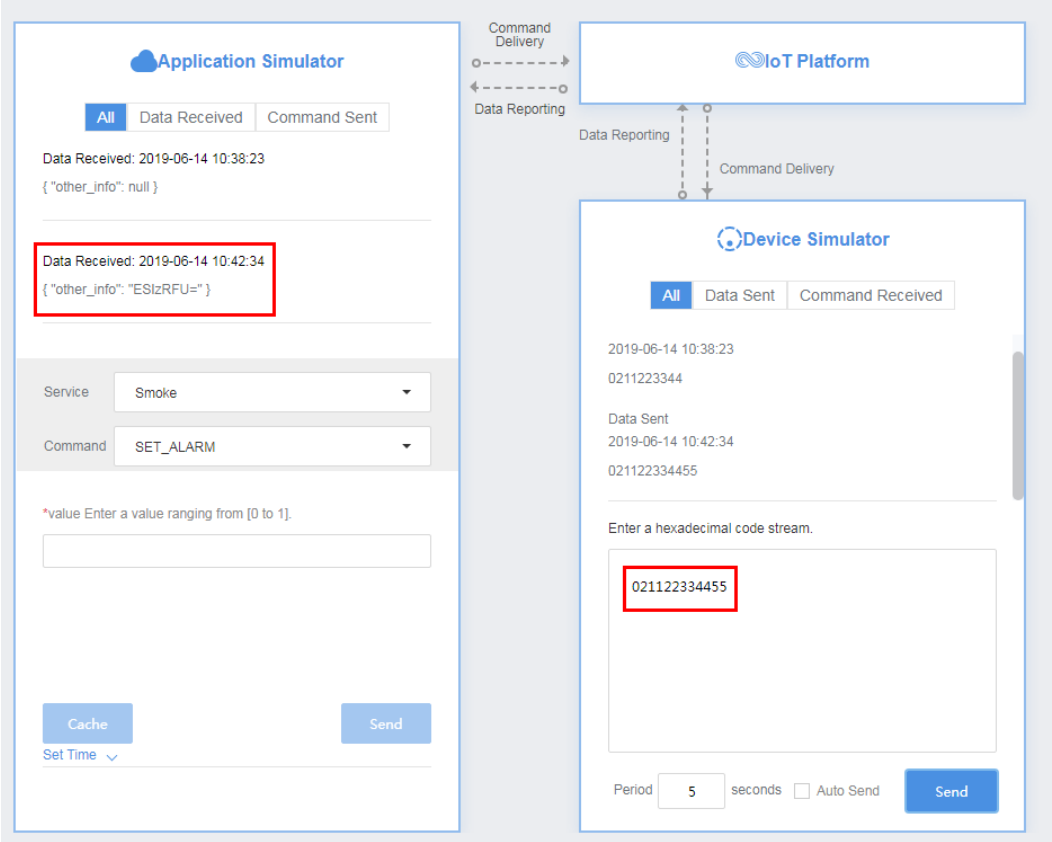
For example, a hexadecimal code stream (0211223344) is reported. In this code stream, **02** indicates the **messageId** field and specifies that this message reports the description of the array type. **11223344** indicates the description and its length is four bytes.

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.



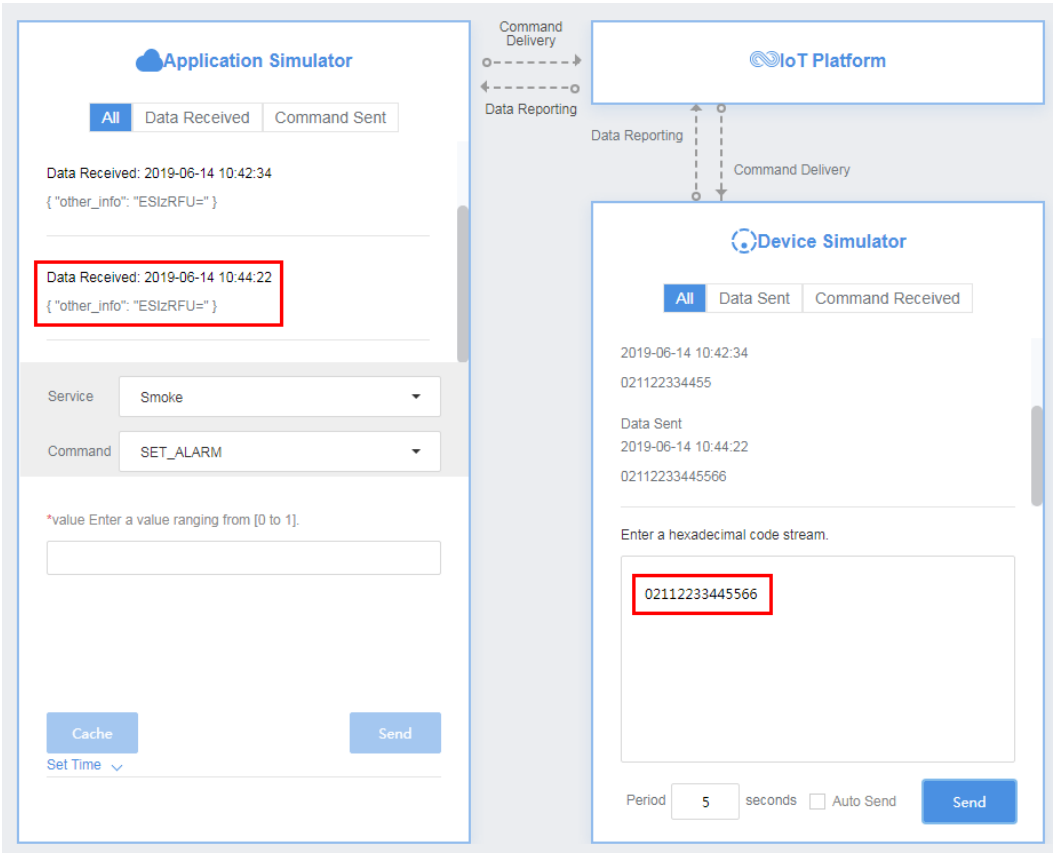
In the second hexadecimal code stream example (021122334455), **02** indicates the **messageId** field and specifies that this message reports the description of the array type. **1122334455** indicates the description and its length is five bytes.

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



In the third 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.

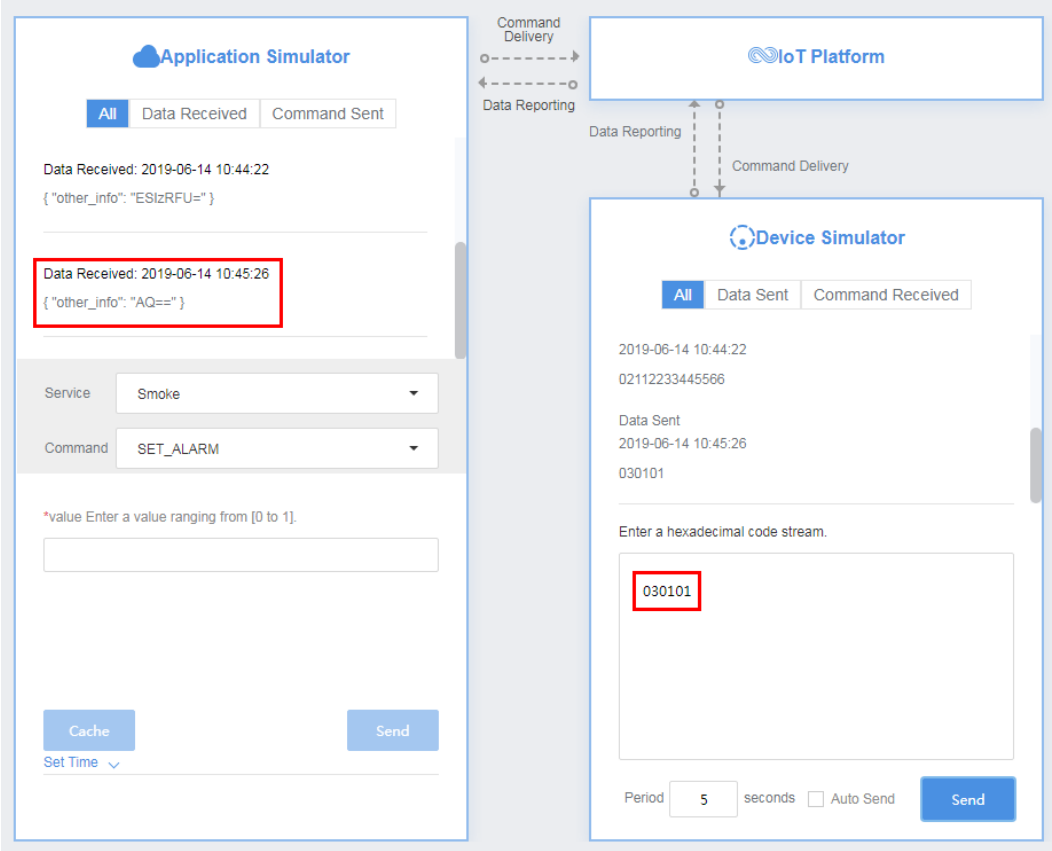
View the data reporting result ({other_info=ESIzRF=}) in **Application Simulator**. The length of the description exceeds six bytes. Therefore, the first five bytes are intercepted and parsed by the codec.



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

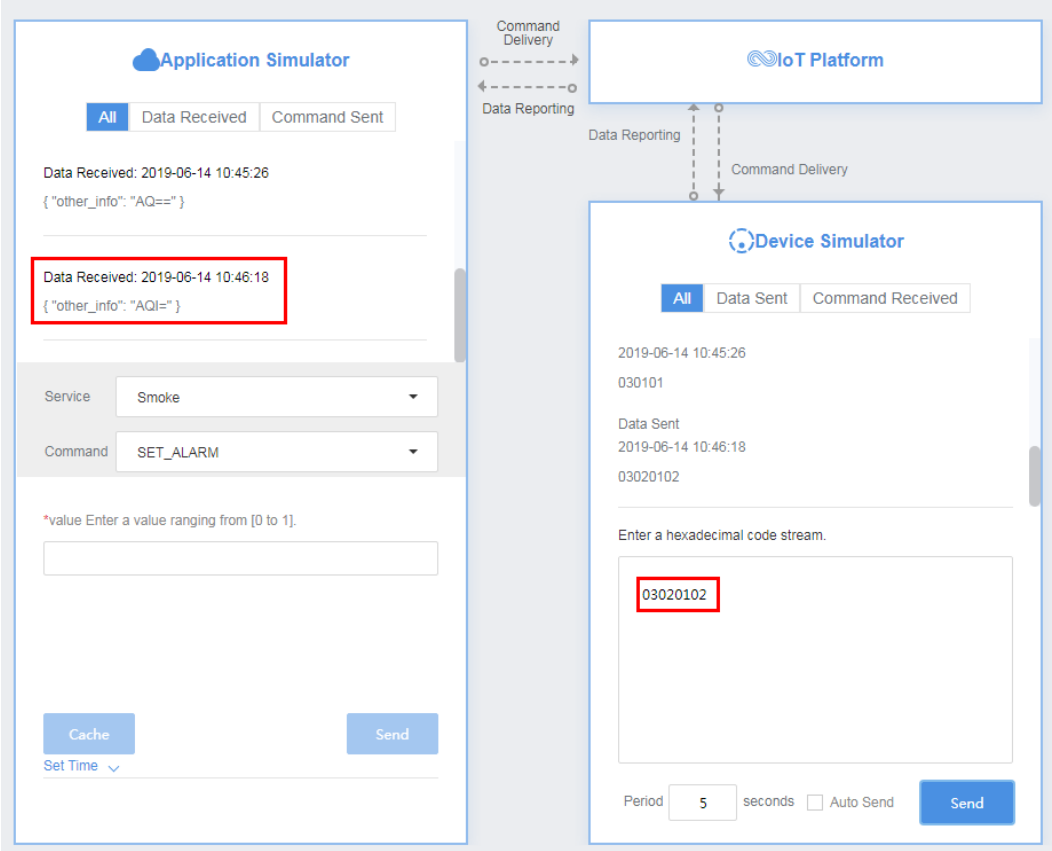
For example, a hexadecimal code stream (030101) is reported. In this code stream, **03** indicates the **messageId** field and specifies that this message reports the description of the variable-length array type. The first **01** indicates the length of the description (one byte) and its length is one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

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



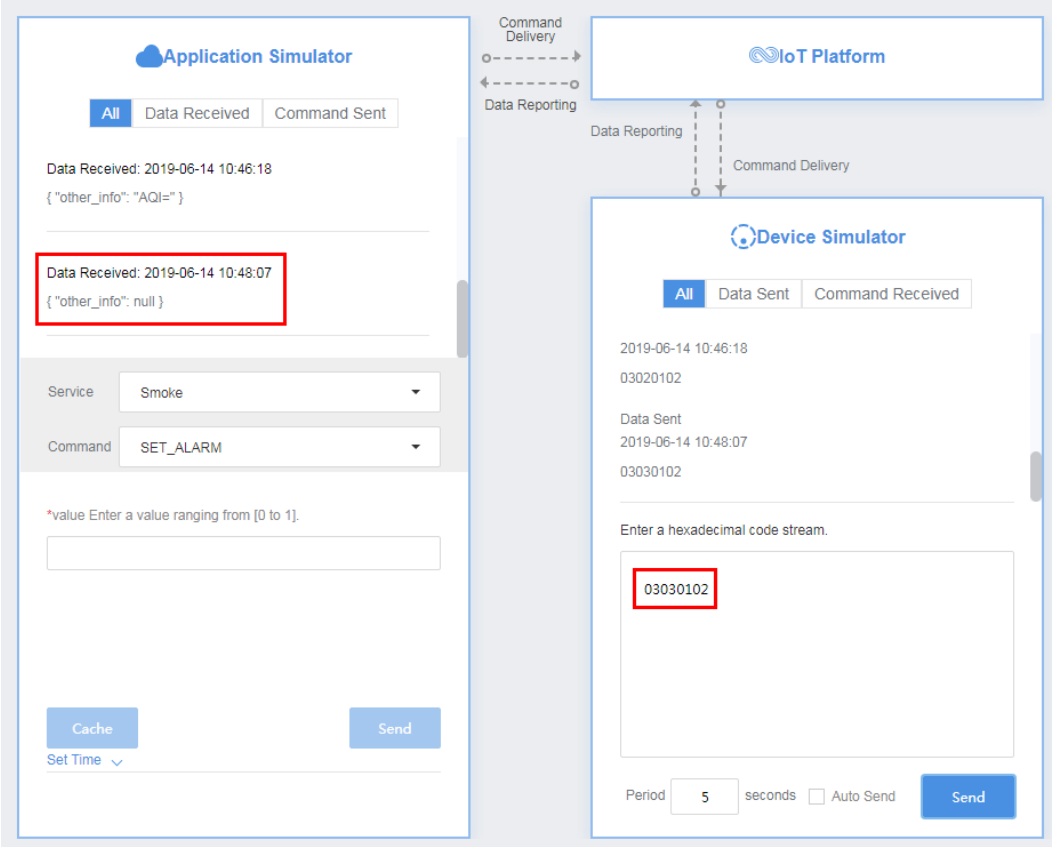
In the second hexadecimal code stream example (03020102), **03** indicates the **messageId** field and specifies that this message reports the description of the variable-length array type. **02** indicates the length of the description (two bytes) and its length is one byte. **0102** indicates the description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

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



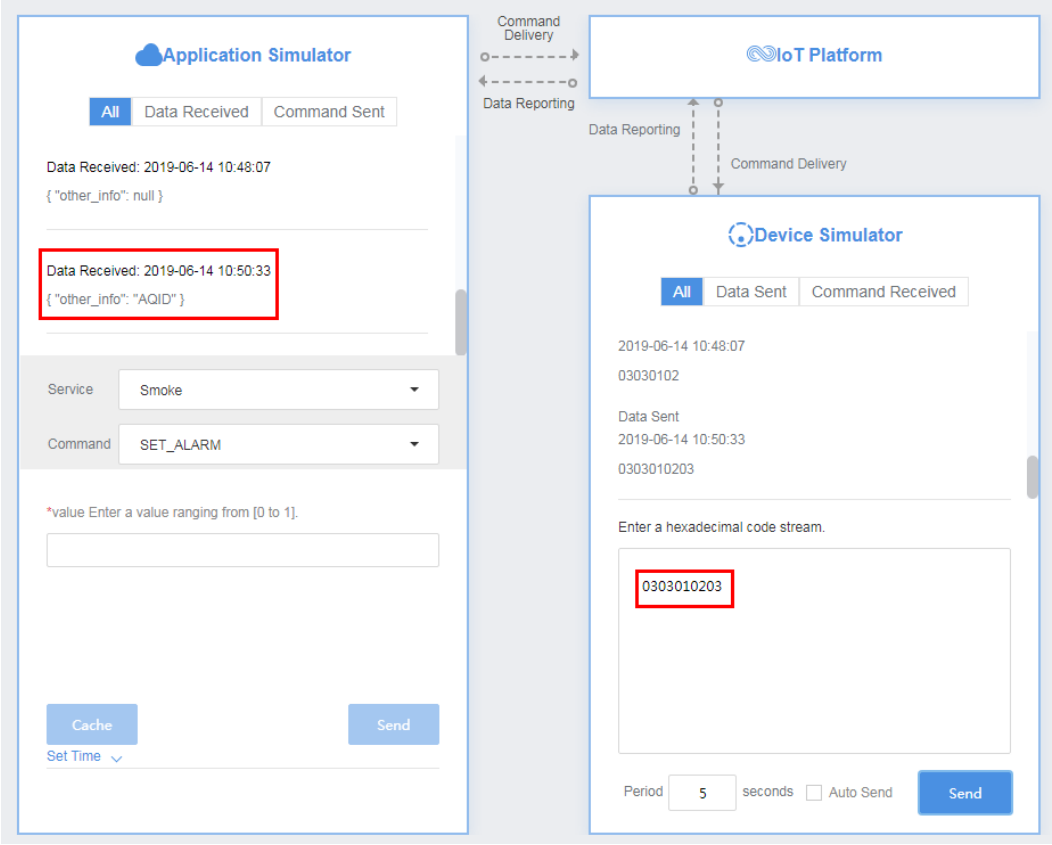
In the third hexadecimal code stream example (03030102), **03** indicates the **messageId** field and specifies that this message reports the description of the variable-length array type. **03** indicates the length of the description (three bytes) and its length is one byte. **0102** indicates the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



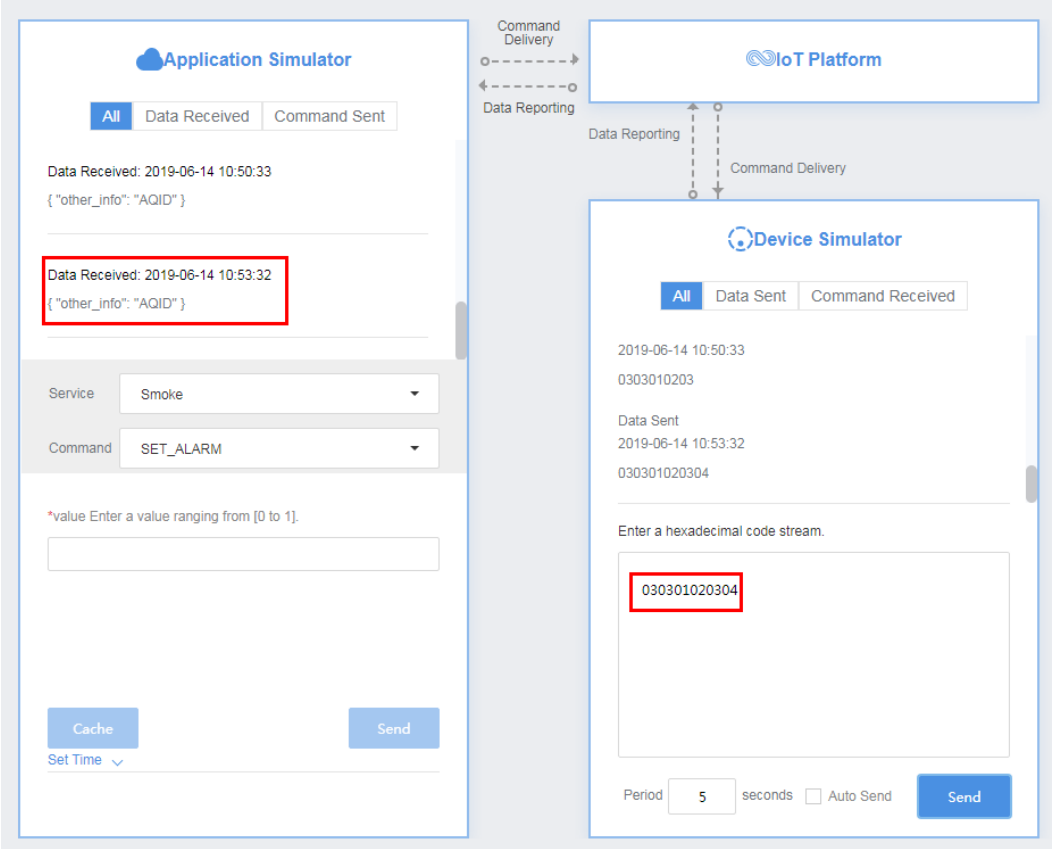
In the fourth hexadecimal code stream example (0303010203), **03** indicates the **messageId** field and specifies that this message reports the description of the variable-length array type. The second **03** indicates the length of the description (three bytes) and its length is one byte. **010203**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

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



In the fifth 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.



----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 to 00000001. Therefore, 100 can be converted to 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 graphically encode data using Base64, hexadecimal code streams are encoded as **values**.

3.4.3 Developing a Codec Using JavaScript

The IoT platform can encode and decode JavaScript scripts. Based on the script files you submit, the IoT platform can convert between binary data and JSON data. 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 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.

Example of a Smoke Detector

Scenario

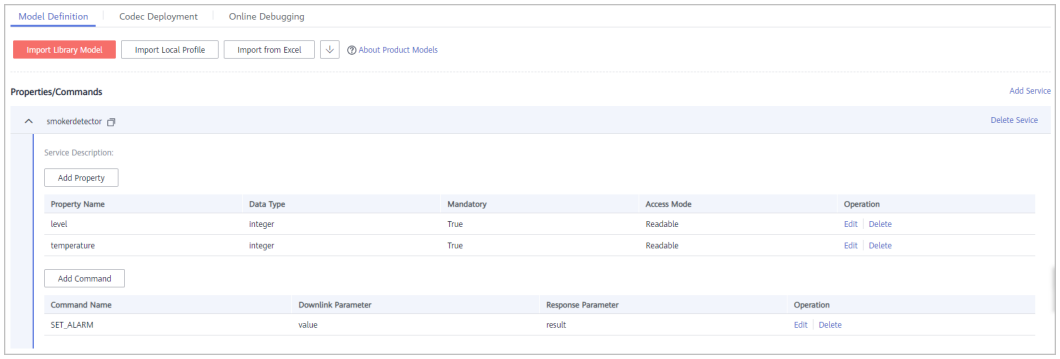
A smoke detector provides the following functions:

- Reporting smoke alarms (fire severity) and temperature
- Remote control commands, which can 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.

Profile Definition

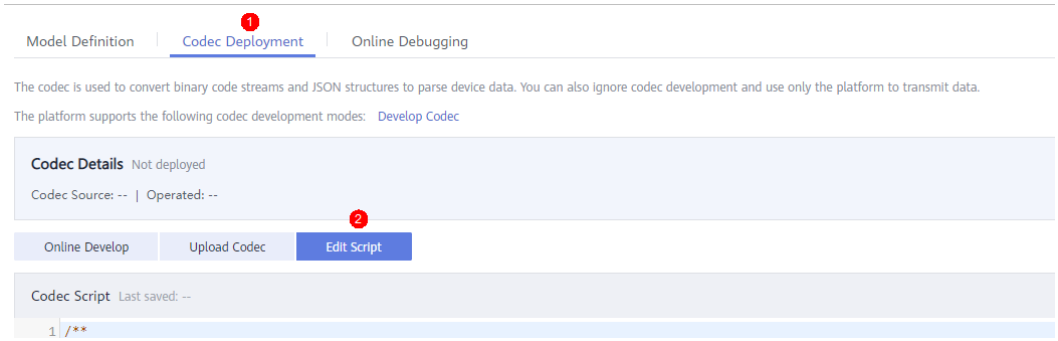
Define the product model on the product details page of the smoke detector.

- **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 is disabled, and the value **1** indicates that the alarm is enabled.



Developing a Codec

Step 1 On the product details page of the smoke detector, select **Codec Development** 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 type
var MSG_TYPE_PROPERTIES_REPORT = 'properties_report'; //Reporting device properties
var MSG_TYPE_COMMAND_RSP = 'command_response'; //Returning a command response
var MSG_TYPE_PROPERTIES_SET_RSP = 'properties_set_response'; //Returning a response for property setting
var MSG_TYPE_PROPERTIES_GET_RSP = 'properties_get_response'; //Returning a response for property query
var MSG_TYPE_MESSAGE_UP = 'message_up'; //Reporting device messages
//Downstream message type
Command Delivery from the var MSG_TYPE_COMMANDS = 'commands'; //Delivering a command
var MSG_TYPE_PROPERTIES_SET = 'properties_set'; //Delivering a property setting request
 varMSG_TYPE_PROPERTIES_GET='properties_get';//Delivering a property query request
var MSG_TYPE_MESSAGE_DOWN = 'messages'; //Delivering platform messages
//Mapping between topics and message types for upstream messages sent by devices
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 decodes 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 parameters:
  {"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 parameters:
  {"msg_type":"command_response","result_code":
0,"command_name":"SET_ALARM","service_id":"smokerdector","paras":{"value":"1"}}
*/
function decode(payload, topic) {
    var jsonObj = {};
    var msgType = '';
   //If the topic parameter exists, parse the message type based on the topic parameter.
    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 to 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 data to the JSON format used by property reporting.
        jsonObj = {"msg_type":"properties_report","services":[{"service_id":serviceId,"properties":
{"level":level,"temperature":temperature}}]};
    }else if (msgType == MSG_TYPE_COMMAND_RSP) { //Convert binary data to the format used by a
command response.
        //Set the value of serviceId. The value corresponds to smokerdector in the product model.
        var serviceId = 'smokerdector';
        var command = dataView.getInt8(0); //Obtain the command name ID from the binary code stream.
```



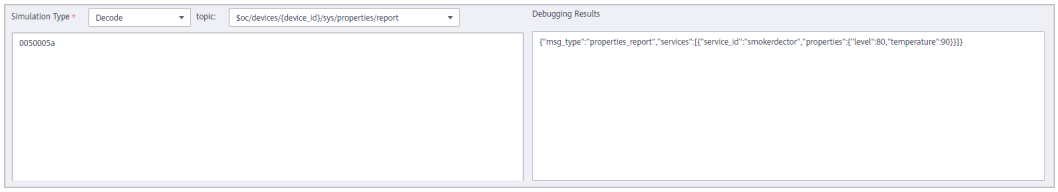
```
        var command_name = '';
        if (2 == command) {
            command_name = 'SET_ALARM';
        }
        var result_code  = dataView.getInt16(1); //Obtain the command execution result from the binary code
stream.
        var value = dataView.getInt8(3); //Obtain the returned value of the command execution result from
the binary code stream.
       //Convert data to the JSON format used by the command response.
        jsonObj =
{"msg_type":"command_response","result_code":result_code,"command_name":command_name,"service_id":
serviceId,"paras":{"value":value}};
    }
    //Convert data to a character string in JSON format.
    return JSON.stringify(jsonObj);
}
/*
Sample data: When a command is delivered, data in JSON format on the platform 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 parameters->
    [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)); //Identifies the command delivery.
        if (jsonObj.command_name == 'SET_ALARM') {
            payload = payload.concat(buffer_uint8(0)); //Indicates the 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.
```



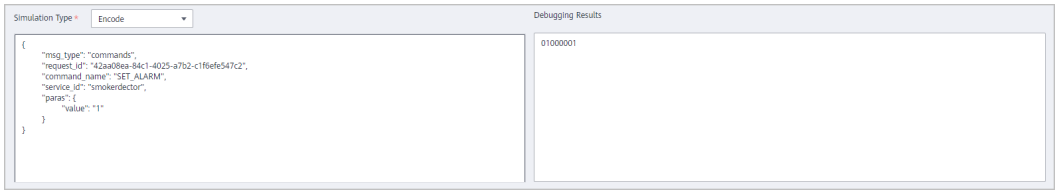
```
function buffer_int32(value) {
    var uint8Array = new Uint8Array(4);
    var dataView = new DataView(uint8Array.buffer);
    dataView.setInt32(0, value);
    return [].slice.call(uint8Array);
}
```

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 reported by the device: $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.

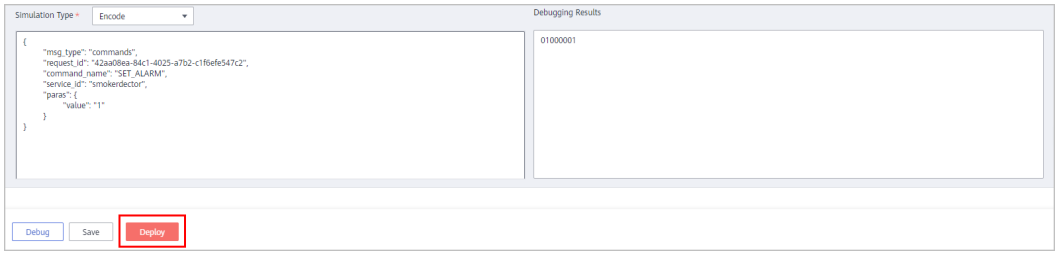
     

   - Check whether the debugging result meets the expectation. If the debugging result does not meet the expectation, modify the code and perform debugging again.
2. Convert a command delivered by an application into binary code streams that can be identified by the device.
   - Select **Encode** for **Simulation 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 original 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 type
var MSG_TYPE_PROPERTIES_REPORT = 'properties_report'; //Reporting device properties
The var MSG_TYPE_COMMAND_RSP = 'command_response'; //Returning a command response
The var MSG_TYPE_PROPERTIES_SET_RSP = 'properties_set_response'; //Returning a property setting response
var MSG_TYPE_PROPERTIES_GET_RSP = 'properties_get_response'; //Returning a property query response
var MSG_TYPE_MESSAGE_UP = 'message_up'; //Reporting message devices
```



```
//Downstream message type
var MSG_TYPE_COMMANDS = 'commands'; //Delivering a command
var MSG_TYPE_PROPERTIES_SET = 'properties_set'; //Delivering a property setting request
 varMSG_TYPE_PROPERTIES_GET='properties_get';//Delivering a property query request
var MSG_TYPE_MESSAGE_DOWN = 'messages'; //Delivering platform messages
//Mapping between topics and message types for upstream messages sent by devices
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 parameters:
  {"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 parameters:
  {"msg_type":"command_response","result_code":
0,"command_name":"SET_ALARM","service_id":"smokerdector","paras":{"value":"1"}}
*/
function decode(payload, topic) {
    var jsonObj = {};
    var msgType = '';
   //If the topic parameter exists, parse the message type based on the topic parameter.
    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 to 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 to 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 to 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 dataon 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 parameters->
    [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)); //Identifies the 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 type
var MSG_TYPE_PROPERTIES_REPORT = 'properties_report'; //Reporting device properties
var MSG_TYPE_COMMAND_RSP = 'command_response'; //Returning a command response
//Downstream message type
var MSG_TYPE_COMMANDS = 'commands'; //Delivering a command
var MSG_TYPE_PROPERTIES_REPORT_REPLY = 'properties_report_reply'; //Response for property reporting
//Message type list
var MSG_TYPE_LIST = {
    0: MSG_TYPE_PROPERTIES_REPORT,         //In the code stream, 0 indicates that the device property is
reported.
    1: MSG_TYPE_PROPERTIES_REPORT_REPLY, //In the code stream, 1 indicates the response for property
reporting.
    2: MSG_TYPE_COMMANDS,                  //In the code stream, 2 indicates the command delivery from the
platform.
    3: MSG_TYPE_COMMAND_RSP                //In the code stream, 3 indicates the 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 parameters:
  {"msg_type":"properties_report","services":[{"service_id":"smokerdector","properties":{"level":
80,"temperature":90}}]}
Input parameters:
  payload: [0x03, 0x01, 0x00, 0x00, 0x01]
Output parameters:
```



```
  {"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 to 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 the 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 the platform 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 parameters->
    [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 parameters->
    [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)); //Identifies the 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 encode method. The decode method of the script needs to decode the data to the JSON format defined in the product model of the platform. 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 or Optional	Type	Description
msg_type	Mandatory	String	Indicates the message type. The value is fixed at **properties_report**.
services	Mandatory	List<ServiceProperty>	Indicates a list of device services. For details, see ServiceProperty structure.

ServiceProperty structure

Field	Mandatory or Optional	Type	Description
service_id	Mandatory	String	Identifies a service of the device.
properties	Mandatory	Object	Indicates service properties, which are defined in the product model associated with the device.
event_time	Optional	String	Indicates the UTC time when the device collects data. The format is yyyyMMddTHHmmssZ, for example, **20161219T114920Z**. If this parameter is not carried in the reported data or is in incorrect format, the time when the platform receives the data is used.

- 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 or Optional	Type	Description
msg_type	Mandatory	String	Indicates the message type. The value is fixed at **properties_set_response**. properties_set_response
request_id	Optional	String	Uniquely identifies a request. If this parameter is carried in a message received by a device, the parameter value needs to be carried in the response message 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	Optional	Integer	Indicates the command execution result. **0** indicates an execution success, whereas other values indicate an execution failure. If this parameter is not carried, the execution is considered to be successful.
result_desc	Optional	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 or Optional	Type	Description
msg_type	Mandatory	String	The value is fixed at **properties_get_response**.
request_id	Optional	String	Uniquely identifies a request. If this parameter is carried in a message received by a device, the parameter value needs to be carried in the response message sent to the platform. If the decoded message does not contain this field, the value of **request_id** in the topic is used.
services	Mandatory	List<ServiceProperty>	Indicates a list of device services. For details, see ServiceProperty structure.

ServiceProperty structure

Field	Mandatory or Optional	Type	Description
service_id	Mandatory	String	Identifies a service of the device.
properties	Mandatory	Object	Indicates service properties, which are defined in the product model associated with the device.
event_time	Optional	String	Indicates the UTC time when the device collects data. The format is yyyyMMddTHHmmssZ, for example, **20161219T114920Z**. If this parameter is not carried in the reported data or is in incorrect format, the time when the platform receives the data is used.

- 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 or Optional	Type	Description
msg_type	Mandatory	String	The value is fixed at **command_response**.
request_id	Optional	String	Uniquely identifies a request. If this parameter is carried in a message received by a device, the parameter value needs to be carried in the response message 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	Optional	Integer	Indicates the command execution result. **0** indicates an execution success, whereas other values indicate an execution failure. If this parameter is not carried, the execution is considered to be successful.
response_name	Optional	String	Indicates the response name, which is defined in the product model associated with the device.
paras	Optional	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 or Optional	Type	Description
msg_type	Mandatory	String	The value is fixed at **message_up**.
content	Optional	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 using the encode method. The encode method needs 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 or Optional	Type	Description
msg_type	Mandatory	String	The value is fixed at **commands**.
request_id	Mandatory	String	Uniquely identifies a request. The ID is delivered to the device through a topic.
service_id	Optional	String	Identifies a service of the device.
command_name	Optional	String	Indicates the device command name, which is defined in the product model associated with the device.
paras	Optional	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 or Optional	Type	Description
msg_type	Mandatory	String	The value is fixed at **properties_set**.
request_id	Mandatory	String	Uniquely identifies a request. If this parameter is carried in a message received by a device, the parameter value needs to be carried in the response message sent to the platform.

services	Mandatory	List<ServiceProperty>	Indicates a list of device service data.

ServiceProperty structure

Field	Mandatory or Optional	Type	Description
service_id	Mandatory	String	Identifies a service of the device.
properties	Mandatory	Object	Service properties, which are defined in the product model.

- Querying device properties

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

Field	Mandatory or Optional	Type	Description
msg_type	Mandatory	String	The value is fixed at **properties_get**.
request_id	Mandatory	String	Uniquely identifies a request. The ID is delivered to the device through a topic.
service_id	Optional	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 or Optional	Type	Description

msg_type	Mandatory	String	The value is fixed at **properties_report_reply**.
request	Optional	byte[]	Binary code stream for property reporting.
result_code	Optional	Integer	Execution result of property reporting.
has_more	Optional	Boolean	Whether a cache command exists.

- Message delivery

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

Field	Mandatory or Optional	Type	Description
msg_type	Mandatory	String	The value is fixed at **messages**.
content	Optional	String	Content of command delivery.

3.4.4 Offline Codec Development

A codec can convert binary messages into JSON messages. The JSON format is defined in the profile. Therefore, before developing a codec, you must define the product model of the device.

Codec demo projects are provided to improve the integration efficiency of offline codec development. You are advised to perform secondary development based on a demo project.

Note: Offline codec development is complex and time-consuming. Therefore, **graphical development** is recommended.

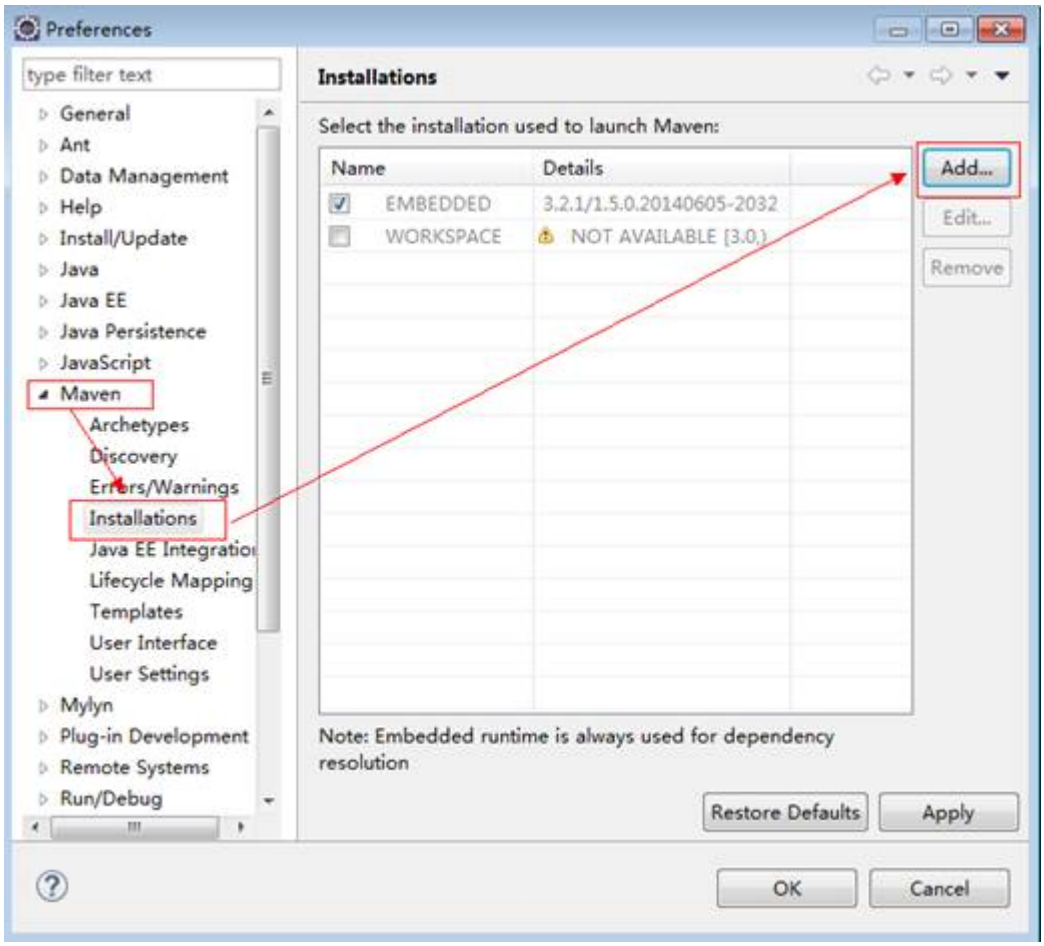
Preparing the Development Environment

- Download the Eclipse installation package from the **official website** and decompress it to a local directory. You can use the software without installation.
- Download the Maven plug-in package (in .zip format) from the **official website** and decompress it to a local directory.
- Install the JDK and configure the Java development environment.

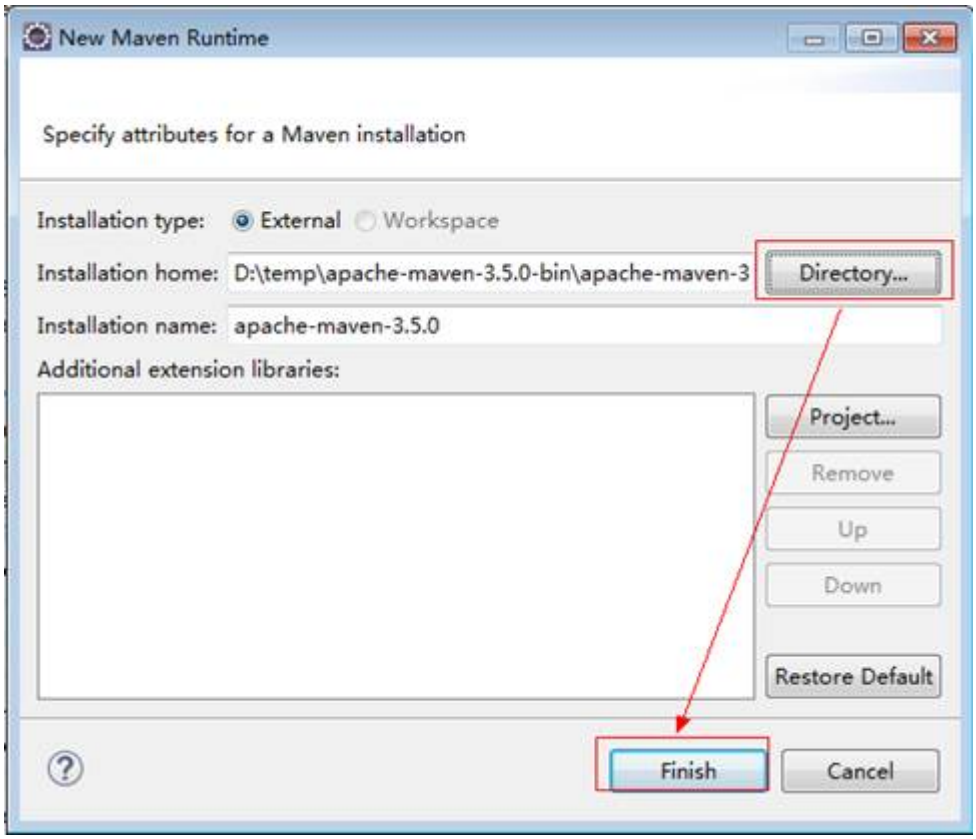
Maven configuration involves setting environment variables on Windows and setting Maven on Eclipse. For details on setting environment variables on

Windows, see other online resources. Maven can be configured on Eclipse as follows:

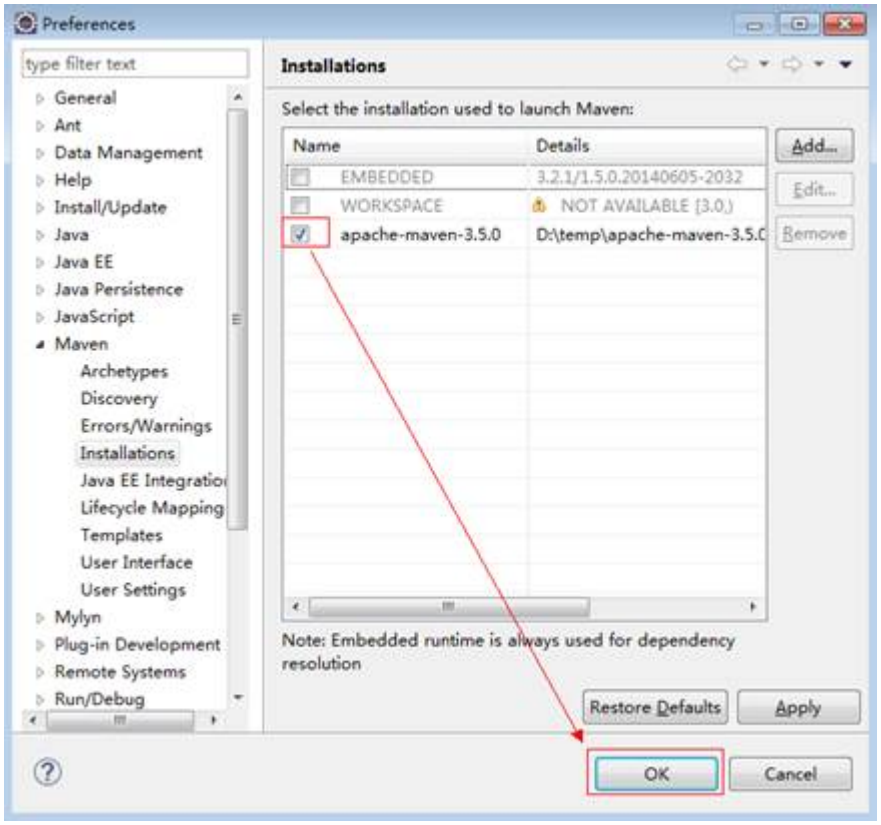
Step 1 Start Eclipse and choose **Windows** > **Preferences**. In the **Preferences** window, choose **Maven** > **Installations**. On the right pane, click **Add**.



Step 2 Select the path where the Maven plug-in package is stored and click **Finish** to import the Maven plug-in.



Step 3 Select the imported Maven plug-in and click **OK**.

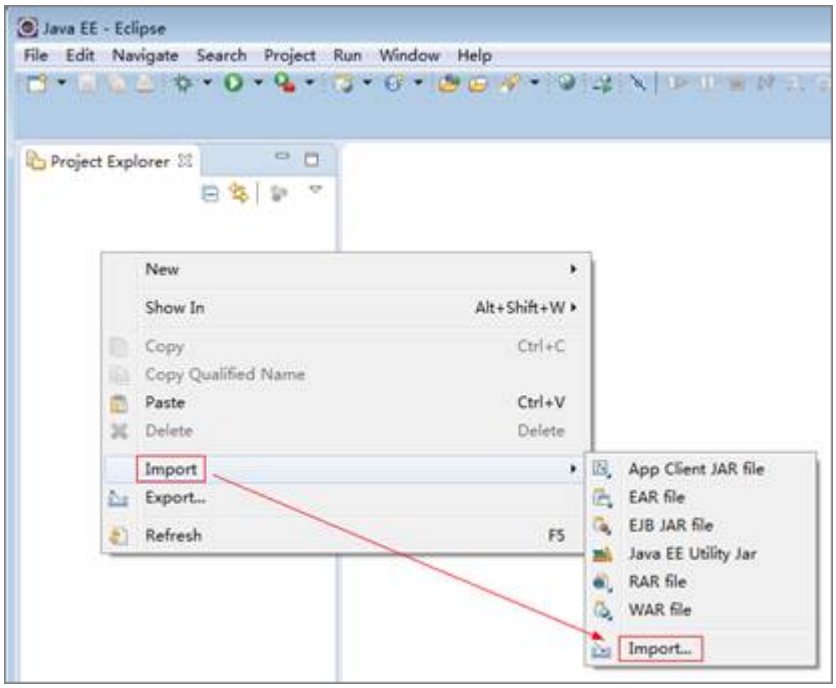


----End

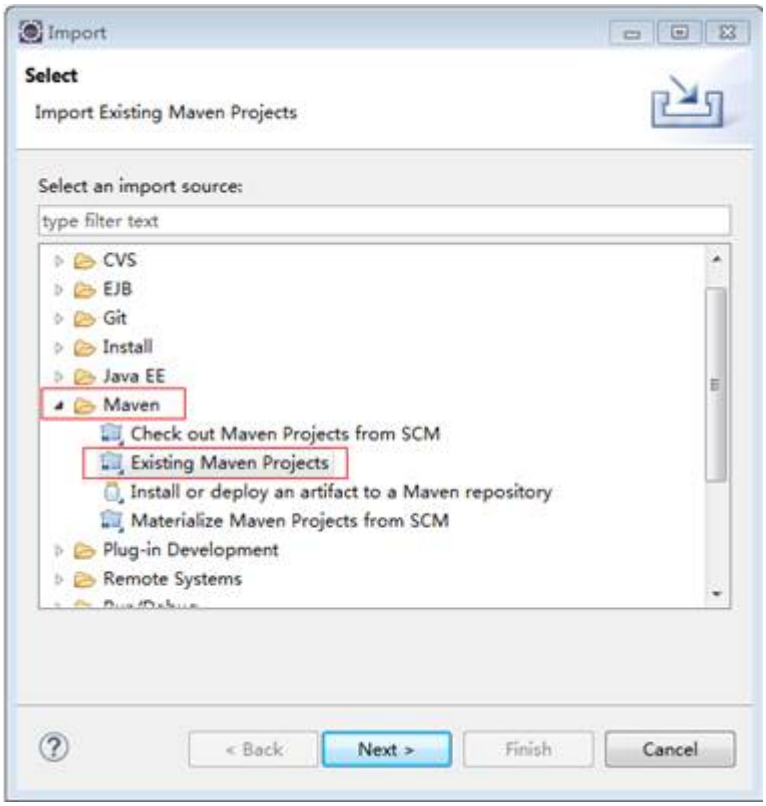
Importing the Demo Project of the Codec

Step 1 Download the demo project, obtain the **codecDemo.zip** file from the **source_code** folder, and decompress the file to a local directory.

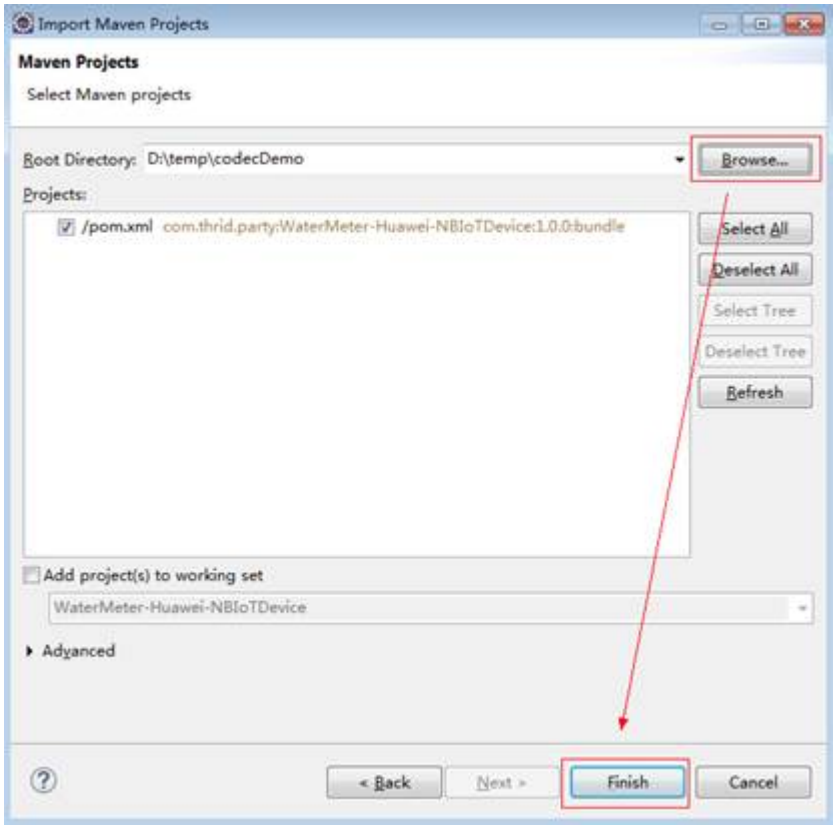
Step 2 Open Eclipse, right-click the blank area in **Project Explorer** on the left of Eclipse, and choose **Import** > **Import...**.



Step 3 Expand **Maven**, select **Existing Maven Projects**, and click **Next**.



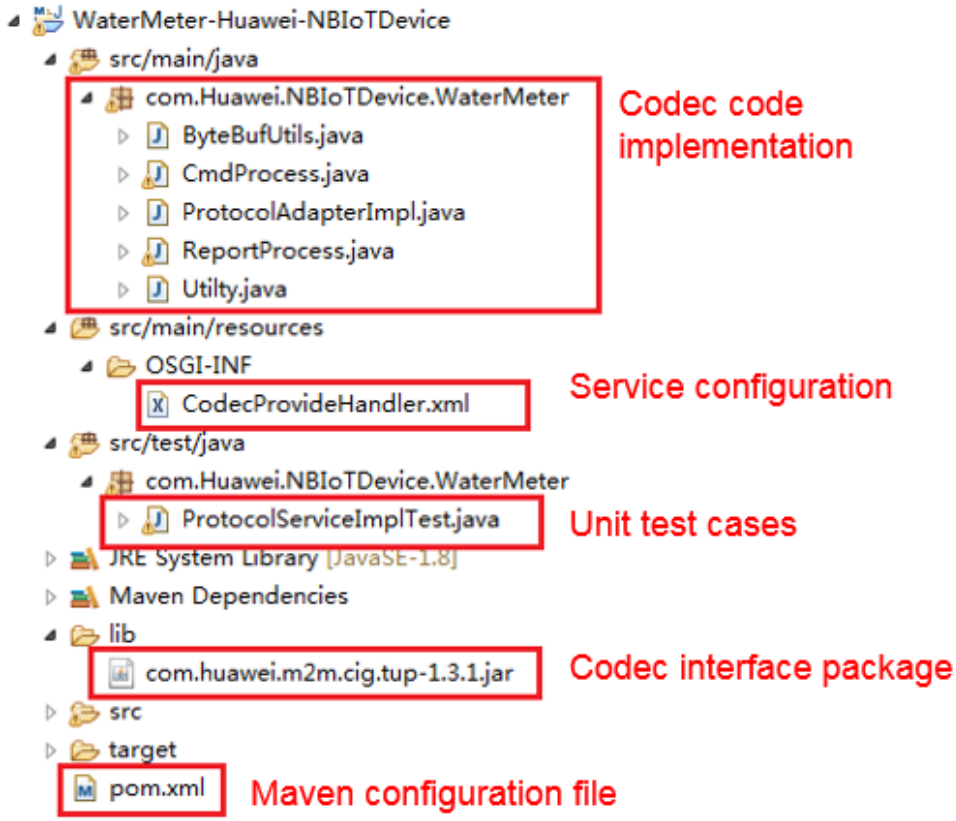
Step 4 Click **Browse**, select the **codecDemo** folder obtained in step 1, select **/pom.xml**, and click **Finish**.



----End

Implementation Sample Interpretation

The following figure shows the structure of the imported codec demo project.



This project is a Maven project. You can modify the following content based on this sample project to obtain the required codec.

Step 1 Modify the configuration files of the Maven project.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<groupId>com.thrid.party</groupId>
<!-- Change it to the name of your codec. The naming rule is as follows: device type-manufacturer ID, for example: WaterMeter-Huawei.-->
<artifactId>WaterMeter-Huawei</artifactId>
<version>1.0.0</version>
<!-- Check that the value is bundle. The value cannot be jar. -->
<packaging>bundle</packaging>

......

<dependencies>
......
   <!-- Codec interface provided by Huawei, which must be introduced. -->
   <!-- Replace systemPath with your local \codecDemo\lib\com.huawei.m2m.cig.tup-1.3.1.jar -->
   <dependency>
```



```
        <groupId>com.huawei</groupId>
        <artifactId>protocal-jar</artifactId>
        <version>1.3.1</version>
        <scope>system</scope>
        <systemPath>${basedir}/lib/com.huawei.m2m.cig.tup-1.3.1.jar</systemPath>
    </dependency>
......
</dependencies>
<build>
<plugins>
    <!-- OSGi packaging configuration -->
    <plugin>
        <configuration>
            <instructions>
                <!-- Change it to the name of your codec. The naming rule is as follows: device type-manufacturer ID, for example: WaterMeter-Huawei. -->
                <Bundle-SymbolicName>WaterMeter-Huawei</Bundle-SymbolicName>
            </instructions>
        </configuration>
    </plugin>
</plugins>
</build>
</project>
```

Step 2 In the **ProtocolAdapterImpl.java** file, change the values of **MANU_FACTURERID**.

```
private static final Logger logger = LoggerFactory.getLogger(ProtocolAdapterImpl.class);
//Manufacturer name
private static final String MANU_FACTURERID = "Huawei";
```

Step 3 Modify the code in the **CmdProcess.java** file so that the codec can encode delivered commands and responses to reported data.

```
package com.Huawei.NBIoTDevice.WaterMeter;

import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.node.ObjectNode;

public class CmdProcess {

    //private String identifier = "123";
    private String msgType = "deviceReq";
    private String serviceId = "Brightness";
    private String cmd = "SET_DEVICE_LEVEL";
    private int hasMore = 0;
    private int errcode = 0;
    private int mid = 0;
    private JsonNode paras;


    public CmdProcess() {
    }

    public CmdProcess(ObjectNode input) {

        try {
            // this.identifier = input.get("identifier").asText();
            this.msgType = input.get("msgType").asText();
            /*
            The IoT platform receives messages reported by the device and encodes the ACK message.
            {
                "identifier":"0",
                "msgType":"cloudRsp",
                "request": ***,//Stream reported by the device
                "errcode":0,
                "hasMore":0
            }
            * */
            if (msgType.equals("cloudRsp")) {
                //Assemble the values of fields in the ACK message.
                this.errcode = input.get("errcode").asInt();
```



```
            this.hasMore = input.get("hasMore").asInt();
        } else {
        /*
        The IoT platform delivers a command to the device with parameters specified as follows:
        {
            "identifier":0,
            "msgType":"cloudReq",
            "serviceId":"WaterMeter",
            "cmd":"SET_DEVICE_LEVEL",
            "paras":{"value":"20"},
            "hasMore":0

        }
        * */
            //Compatibility must be considered. If the MID is not transferred, it is not encoded.
            if (input.get("mid") != null) {
                this.mid = input.get("mid").intValue();
            }
            this.cmd = input.get("cmd").asText();
            this.paras = input.get("paras");
            this.hasMore = input.get("hasMore").asInt();
        }

    } catch (Exception e) {
        e.printStackTrace();
    }

}

public byte[] toByte() {
    try {
        if (this.msgType.equals("cloudReq")) {
            /*
            The NA delivers a control command. In this example, there is only one command:
SET_DEVICE_LEVEL.
            If there are other commands, determine them.
            * */
            if (this.cmd.equals("SET_DEVICE_LEVEL")) {
                int brightlevel = paras.get("value").asInt();
                byte[] byteRead = new byte[5];
                ByteBufUtils buf = new ByteBufUtils(byteRead);
                buf.writeByte((byte) 0xAA);
                buf.writeByte((byte) 0x72);
                buf.writeByte((byte) brightlevel);

               //Compatibility must be considered. If the MID is not transferred, it is not encoded.
                if (Utilty.getInstance().isValidofMid(mid)) {
                    byte[] byteMid = new byte[2];
                    byteMid = Utilty.getInstance().int2Bytes(mid, 2);
                    buf.writeByte(byteMid[0]);
                    buf.writeByte(byteMid[1]);
                }

                return byteRead;
            }
        }

        /*
        After receiving the data reported by the device, the IoT platform encodes the ACK message as
required and responds to the device. If null is returned, the IoT platform does not need to respond.
        * */
        else if (this.msgType.equals("cloudRsp")) {
            byte[] ack = new byte[4];
            ByteBufUtils buf = new ByteBufUtils(ack);
            buf.writeByte((byte) 0xAA);
            buf.writeByte((byte) 0xAA);
            buf.writeByte((byte) this.errcode);
            buf.writeByte((byte) this.hasMore)
            return ack;
```



```
            }
            return null;
        } catch (Exception e) {
            // TODO: handle exception
            e.printStackTrace();
            return null;
        }
    }

}
```

Step 4 Modify the code in the **ReportProcess.java** file so that the codec can decode data reported by devices and command execution results.

```
package com.Huawei.NBIoTDevice.WaterMeter;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.node.ArrayNode;
import com.fasterxml.jackson.databind.node.ObjectNode;

public class ReportProcess {
    //private String identifier;

    private String msgType = "deviceReq";
    private int hasMore = 0;
    private int errcode = 0;
    private byte bDeviceReq = 0x00;
    private byte bDeviceRsp = 0x01;

    //serviceId = Brightness
    private int brightness = 0;

    //serviceId = Electricity
    private double voltage = 0.0;
    private int current = 0;
    private double frequency = 0.0;
    private double powerfactor = 0.0;

    //serviceId = Temperature
    private int temperature = 0;

    private byte noMid = 0x00;
    private byte hasMid = 0x01;
    private boolean isContainMid = false;
    private int mid = 0;

    /**
     * @param binaryData: Payload of the CoAP packet sent by the device to the IoT platform
     *                    Input parameters in this example: AA 72 00 00 32 08 8D 03 20 62 33 99
     *                    byte[0]--byte[1]: AA 72 command header
     *                    byte[2]:   00 mstType: 00 represents deviceReq, which indicates that data is reported by the device.
     *                    byte[3]:   00 hasMore: 0 indicates that there is no subsequent data and 1 indicates that there is subsequent data. If the hasMore field is not contained, the value 0 is used.
     *                    byte[4]--byte[11]: indicates service data, which is parsed as required.//If the service data is deviceRsp, byte[4] indicates whether the MID is carried and byte[5] to byte[6] indicate the short command ID.
     * @return
     */
    public ReportProcess(byte[] binaryData) {
        //The identifier parameter can be obtained based on the input parameter stream. In this example, the default value is 123.
        // identifier = "123";

        /*
        If the data is reported by the device, the return value is in the following format:
        {
            "identifier":"123",
            "msgType":"deviceReq",
            "hasMore":0,
```



```
        "data":[{"serviceId":"Brightness",
                "serviceData":{"brightness":50},
                {
                "serviceId":"Electricity",
                "serviceData":{"voltage":218.9,"current":800,"frequency":50.1,"powerfactor":0.98},
                {
                "serviceId":"Temperature",
                "serviceData":{"temperature":25},
                ]
    }
    */
    if (binaryData[2] == bDeviceReq) {
        msgType = "deviceReq";
        hasMore = binaryData[3];

        //serviceId = Brightness
        brightness = binaryData[4];

        //serviceId = Electricity
        voltage = (double) (((binaryData[5] << 8) + (binaryData[6] & 0xFF)) * 0.1f);
        current = (binaryData[7] << 8) + binaryData[8];
        powerfactor = (double) (binaryData[9] * 0.01);
        frequency = (double) binaryData[10] * 0.1f + 45;

        //serviceId = Temperature
        temperature = (int) binaryData[11] & 0xFF - 128;
    }
    /*
    If the data is a response sent by the device to a command of the IoT platform, the return value is in
the following format:
    {
        "identifier":"123",
        "msgType":"deviceRsp",
        "errcode":0,
        "body" :{****} Note that the body is a JSON structure.
    }
    */
    else if (binaryData[2] == bDeviceRsp) {
        msgType = "deviceRsp";
        errcode = binaryData[3];
        //Compatibility must be considered. If the MID is not transferred, it is not encoded.
        if (binaryData[4] == hasMid) {
            mid = Utilty.getInstance().bytes2Int(binaryData, 5, 2);
            if (Utilty.getInstance().isValidofMid(mid)) {
                isContainMid = true;
            }

        }
    } else {
        return;
    }


}

public ObjectNode toJsonNode() {
    try {
        //Assemble the body.
        ObjectMapper mapper = new ObjectMapper();
        ObjectNode root = mapper.createObjectNode();

        // root.put("identifier", this.identifier);
        root.put("msgType", this.msgType);

        //Assemble the message body based on the msgType field.
        if (this.msgType.equals("deviceReq")) {
            root.put("hasMore", this.hasMore);
            ArrayNode arrynode = mapper.createArrayNode();
```

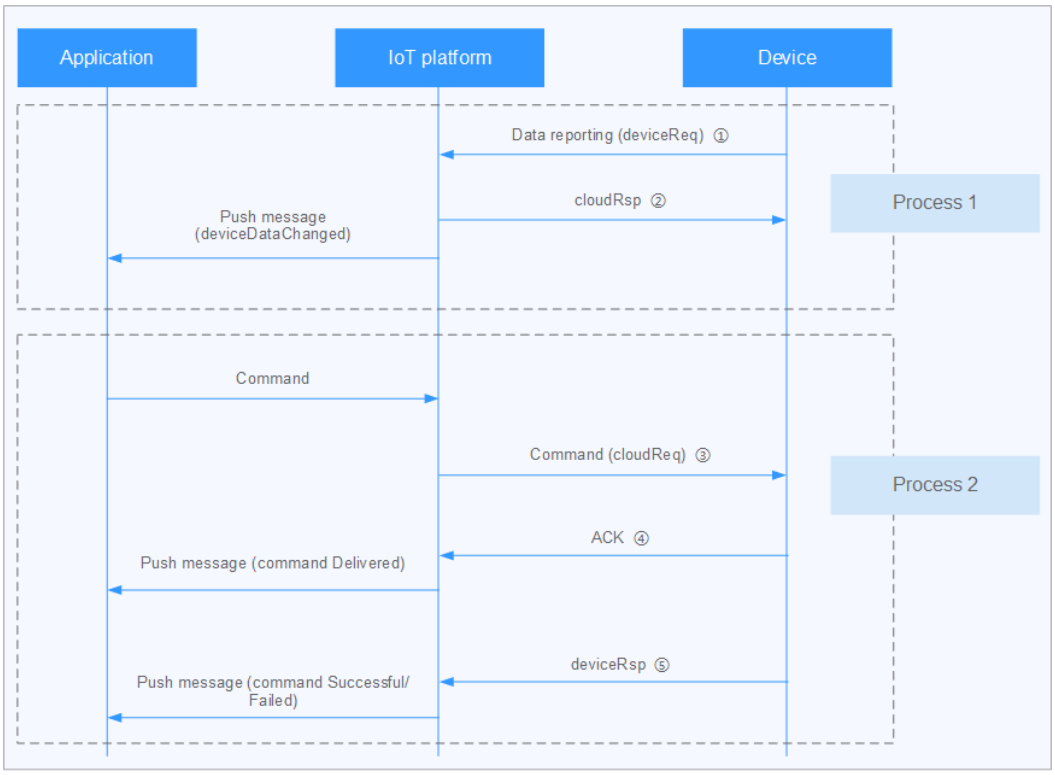


```
            //serviceId = Brightness
            ObjectNode brightNode = mapper.createObjectNode();
            brightNode.put("serviceId", "Brightness");
            ObjectNode brightData = mapper.createObjectNode();
            brightData.put("brightness", this.brightness);
            brightNode.put("serviceData", brightData);
            arrynode.add(brightNode);
            //serviceId = Electricity
            ObjectNode electricityNode = mapper.createObjectNode();
            electricityNode.put("serviceId", "Electricity");
            ObjectNode electricityData = mapper.createObjectNode();
            electricityData.put("voltage", this.voltage);
            electricityData.put("current", this.current);
            electricityData.put("frequency", this.frequency);
            electricityData.put("powerfactor", this.powerfactor);
            electricityNode.put("serviceData", electricityData);
            arrynode.add(electricityNode);
            //serviceId = Temperature
            ObjectNode temperatureNode = mapper.createObjectNode();
            temperatureNode.put("serviceId", "Temperature");
            ObjectNode temperatureData = mapper.createObjectNode();
            temperatureData.put("temperature", this.temperature);
            temperatureNode.put("serviceData", temperatureData);
            arrynode.add(temperatureNode);

            //serviceId = Connectivity
            ObjectNode ConnectivityNode = mapper.createObjectNode();
            ConnectivityNode.put("serviceId", "Connectivity");
            ObjectNode  ConnectivityData = mapper.createObjectNode();
            ConnectivityData.put("signalStrength", 5);
            ConnectivityData.put("linkQuality", 10);
            ConnectivityData.put("cellId", 9);
            ConnectivityNode.put("serviceData", ConnectivityData);
            arrynode.add(ConnectivityNode);

            //serviceId = Battery
            ObjectNode batteryNode = mapper.createObjectNode();
            batteryNode.put("serviceId", "battery");
            ObjectNode batteryData = mapper.createObjectNode();
            batteryData.put("batteryVoltage", 25);
            batteryData.put("battervLevel", 12);
            batteryNode.put("serviceData", batteryData);
            arrynode.add(batteryNode);

            root.put("data", arrynode);

        } else {
            root.put("errcode", this.errcode);
            //Compatibility must be considered. If the MID is not transferred, it is not decoded.
            if (isContainMid) {
                root.put("mid", this.mid);//mid
            }
            //Assemble the body. The body must be an ObjectNode object.
            ObjectNode body = mapper.createObjectNode();
            body.put("result", 0);
            root.put("body", body);
        }
        return root;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
  }
}
```

----End

Description of decode API

The input parameter **binaryData** over the decode API is the payload in the CoAP message sent by a device.



Upstream messages reported by the device need to be processed by the codec in the following two scenarios (message **(4)** is the protocol ACK message returned by the module and does not need to be processed by the codec):

- Reported device data (message **(1)** in the figure)

Parameter	Type	Mandatory or Optional	Description
identifier	String	No	Identifier of the device in the application protocol. The IoT platform obtains the parameter over the decode API, encodes the parameter over the encode API, and places the parameter in a stream.
msgType	String	Yes	This parameter has a fixed value of **deviceReq**, which indicates that the device reports data to the IoT platform.

Parameter	Type	Mandatory or Optional	Description
hasMore	Int	No	Specifies whether the IoT platform has subsequent commands to deliver. **0**: The IoT platform does not have subsequent commands to deliver. **1**: The IoT platform has subsequent commands to deliver. Subsequent data indicates that a piece of data reported by a device may be reported multiple times. After the data is reported the current time, the IoT platform determines whether there are subsequent messages using the **hasMore** field. The **hasMore** field is valid only in PSM mode. When the **hasMore** field of reported data is set to **1**, the IoT platform does not deliver cached commands until it receives reported data whose **hasMore** field is set to **0**. If the reported data does not contain the **hasMore** field, the IoT platform processes the data on the basis that the **hasMore** field is set to **0**.
data	ArrayNode	Yes	Content of the data reported by the device.

Table 3-1 Definition of ArrayNode

Parameter	Type	Mandatory or Optional	Description
serviceId	String	Yes	Service ID.
serviceData	ObjectNode	Yes	Data of a service. The detailed parameters are defined in the profile.
eventTime	String	No	Specifies the time when the device collects data. The format is yyyyMMddTHHmmssZ, for example, 20161219T114920Z.

Example:


```
{
   "identifier": "123",
   "msgType": "deviceReq",
   "hasMore": 0,
   "data": [{
   "serviceId": "NBWaterMeterCommon",
   "serviceData": {
      "meterId": "xxxx",
      "dailyActivityTime": 120,
      "flow": "565656",
      "cellId": "5656",
      "signalStrength": "99",
      "batteryVoltage": "3.5"
   },
   "eventTime": "20160503T121540Z"
   },
   {
   "serviceId": "waterMeter",
   "serviceData": {
      "internalTemperature": 256
   },
   "eventTime": "20160503T121540Z"
   }]
}
```

- Device response to the command delivered by the IoT platform (message **(5)** in the figure)

Parameter	Type	Description	Mandatory or Optional
identifier	String	Identifier of the device in the application protocol. The IoT platform obtains the parameter over the decode API, encodes the parameter over the encode API, and places the parameter in a stream.	No
msgType	String	This parameter has a fixed value of **deviceRsp**, which indicates a response sent by a device to the IoT platform.	Yes

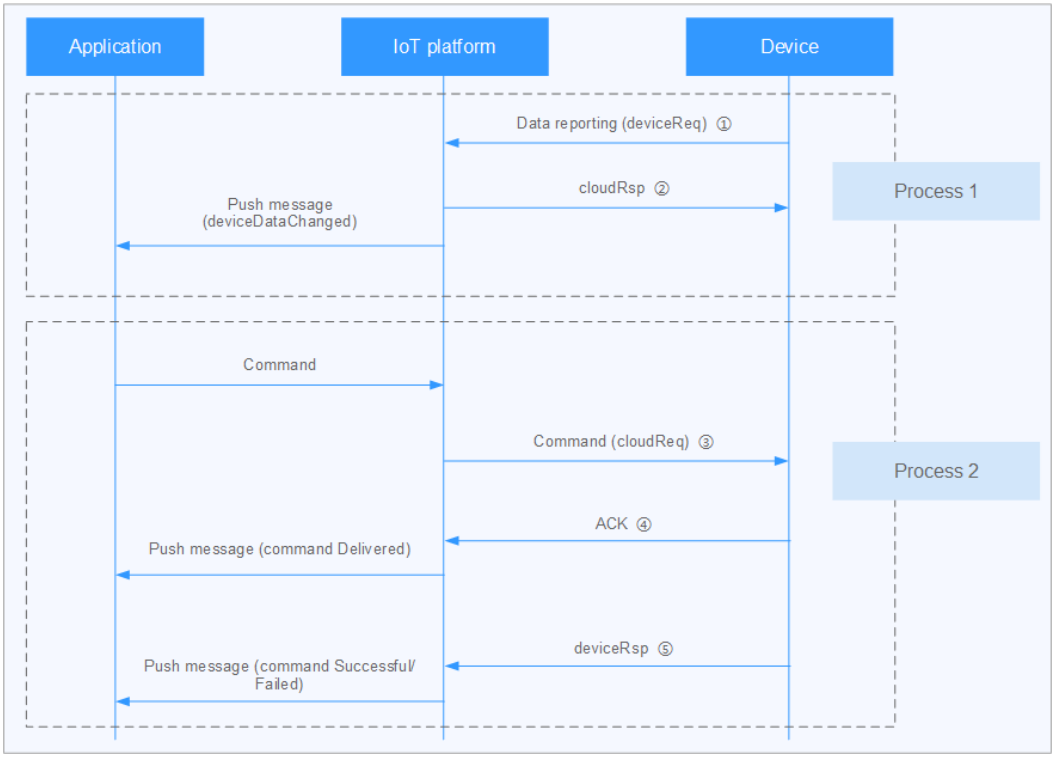
Parameter	Type	Description	Mandatory or Optional
mid	Int	Specifies a 2-byte unsigned command ID. If the device must return the command execution result (deviceRsp), this field is used to associate the command execution result (deviceRsp) with the corresponding command. When the IoT platform delivers a command over the encode API, the IoT platform places the MID allocated by the IoT platform into a stream and delivers the stream to the device together with the command. When the device reports the command execution result (deviceRsp), the device returns the MID to the IoT platform. In this way, the IoT platform associates the delivered command with the command execution result (deviceRsp) and updates the command delivery status accordingly.	Yes
errcode	Int	Request processing result code. The IoT platform determines the command delivery status based on this field. The value **0** indicates success, and the value **1** indicates failure.	Yes
body	ObjectNode	Command response, whose fields are defined in the profile. **Note:** The body is not an array.	No

Example:

```
{
   "identifier": "123",
   "msgType": "deviceRsp",
   "mid": 2016,
   "errcode": 0,
   "body": {
      "result": 0
   }
}
```

Description of encode API

Input parameters of the encode API are commands or responses in JSON format delivered by the IoT platform.



The downstream messages of the IoT platform can be classified into two types:

- Response from the IoT platform to the data reported by the device (message **(2)** in the figure)

  Table 3-2 Definition of input parameters of the encode API over which the IoT platform responds to data reported by a device

Parameter	Type	Description	Mandatory or Optional
identifier	String	Identifier of the device in the application protocol. The IoT platform obtains the parameter over the decode API, encodes the parameter over the encode API, and places the parameter in a stream.	No
msgType	String	This field has a fixed value of **cloudRsp**, which indicates that the IoT platform sends a response to data reported by a device.	Yes
request	byte[]	Indicates the data reported by the device.	Yes

Parameter	Type	Description	Mandatory or Optional
errcode	int	Request processing result code. The IoT platform determines the command delivery status based on this field. The value **0** indicates success, and the value **1** indicates failure.	Yes
hasMore	int	Specifies whether the IoT platform has subsequent messages to be sent. The value **0** indicates that the IoT platform does not have subsequent messages to be sent. The value **1** indicates that the IoT platform has subsequent messages to be sent. Subsequent messages indicate that the IoT platform still needs to deliver commands, and the **hasMore** field is used to tell the device not to sleep. The **hasMore** field is valid only in PSM mode with the downstream message indication function enabled.	Yes

Note: If **msgType** is set to **cloudRsp** and **null** is returned by the codec detection tool, the codec does not define the response to the reported data and the IoT platform does not need to respond.

Example:

```
{
    "identifier": "123",
    "msgType": "cloudRsp",
    "request": [
        1,
        2
    ],
    "errcode": 0,
    "hasMore": 0
}
```

- Commands delivered by the IoT platform (message **(3)** in the figure)

Table 3-3 Definition of input parameters of the encode API over which the IoT platform delivers commands

Parameter	Type	Description	Mandatory or Optional
identifier	String	Identifier of the device in the application protocol. The IoT platform obtains the parameter over the decode API, encodes the parameter over the encode API, and places the parameter in a stream.	No
msgType	String	This parameter has a fixed value of **cloudReq**, which indicates a command delivered by the IoT platform.	Yes
serviceId	String	Service ID.	Yes
cmd	String	Command name. For details, see the profile.	Yes
paras	ObjectNode	Command parameters, which are defined in the profile.	Yes
hasMore	Int	Specifies whether the IoT platform has subsequent commands to deliver. **0**: The IoT platform does not have subsequent commands to deliver. **1**: The IoT platform has subsequent commands to deliver. Subsequent commands indicate that the IoT platform still needs to deliver commands, and the **hasMore** field is used to tell the device not to sleep. The **hasMore** field is valid only in PSM mode with the downstream message indication function enabled.	Yes

Parameter	Type	Description	Mandatory or Optional
mid	Int	A 2-byte unsigned command ID that is allocated by the IoT platform. (The value ranges from 1 to 65535.) When the IoT platform delivers a command over the encode API, the IoT platform places the MID allocated by the IoT platform into a stream and delivers the stream to the device together with the command. When the device reports the command execution result (deviceRsp), the device returns the MID to the IoT platform. In this way, the IoT platform associates the delivered command with the command execution result (deviceRsp) and updates the command delivery status accordingly.	Yes

Example:

```
{
    "identifier": "123",
    "msgType": "cloudReq",
    "serviceId": "NBWaterMeterCommon",
    "mid": 2016,
    "cmd": "SET_TEMPERATURE_READ_PERIOD",
    "paras": {
        "value": 4
    },
    "hasMore": 0}
}
```

Description of getManufacturerId API

This API is used to return the manufacturer ID in the format of a character string. The IoT platform calls this API to obtain the manufacturer ID.

Example:

```
@Override
public String getManufacturerId() {
    return "TestUtf8ManuId";
}
```

Precautions on Interface Implementation

Support for Thread Security Required

The decode and encode functions must ensure thread security. Therefore, member or static variables cannot be added to cache intermediate data.

- Incorrect example: When multiple threads are started at the same time, the status of thread A is set to **Failed** while the status of thread B is set to **Success**. As a result, the status is incorrect, and the program running is abnormal.

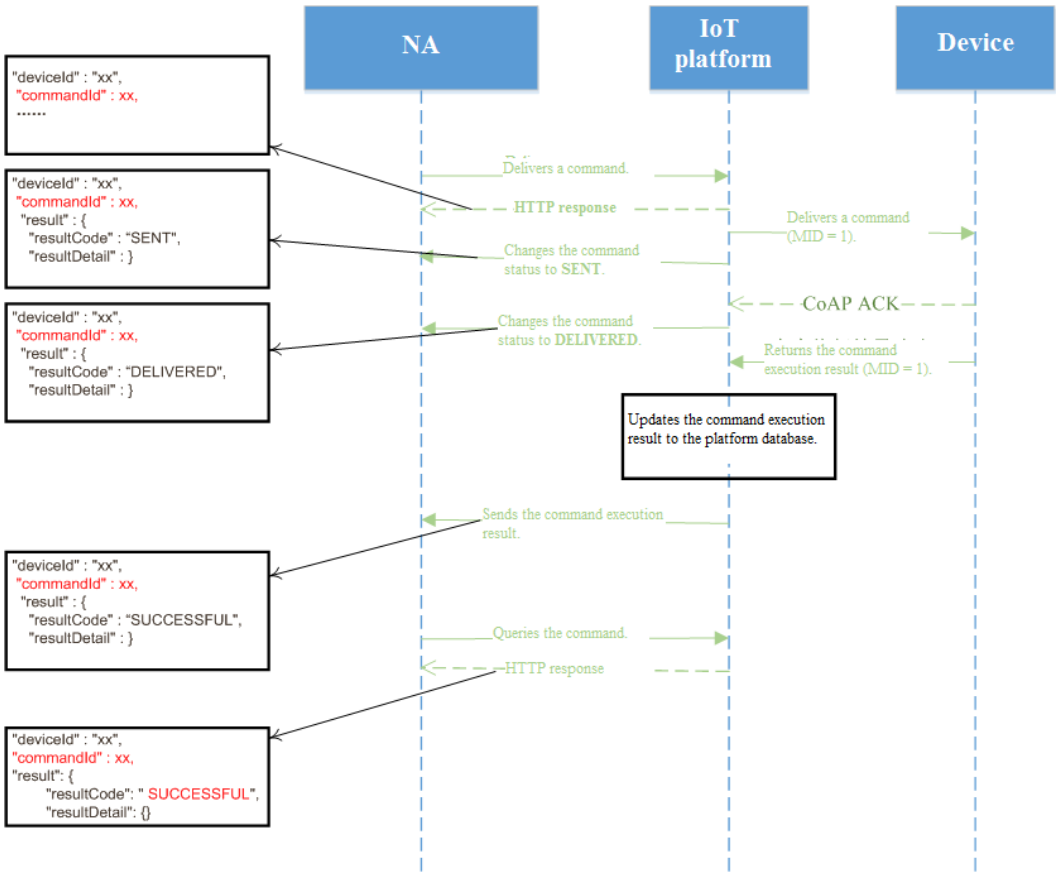
```
public class ProtocolAdapter {
private String status;

@Override
public ObjectNode decode(finalbyte[] binaryData) throws Exception {
    if (binaryData == null) {
        status = "Failed";
        return null;
    }
    ObjectNode node;
    ...;
    status = "Success";//The thread is insecure.
    return node;
}
}
```

- Correct example: Encoding and decoding are performed based on the input parameters, and the encoding and decoding library does not process services.

Explanation of the mid Field

The IoT platform delivers orders in sequence. However, the IoT platform does not respond to the order execution results in the same sequence as the delivered orders. The MID is used to associate the order execution result response with the delivered order. On the IoT platform, whether the MID is implemented affects the message flow.

- **When the MID is implemented:**



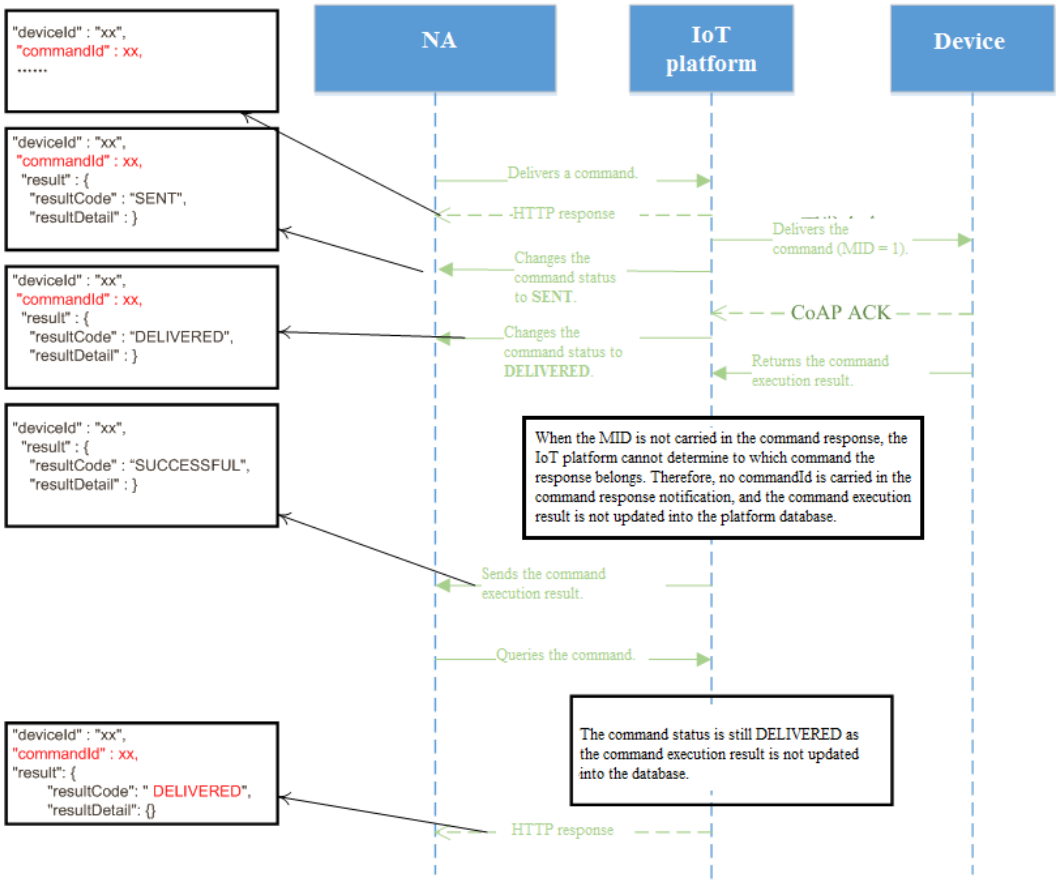
If the MID is implemented and the order execution result is reported successfully:

a. The status (**SUCCESSFUL/FAILED**) in the order execution result is updated to the record of the order in the IoT platform database.

b. The order execution result notification sent by the IoT platform to the NA server contains **commandId**.

c. The query result of the NA server indicates that the status of the order is **SUCCESSFUL/FAILED**.

- **When the MID is not implemented:**



If the MID is not implemented and the order execution result is reported successfully:

a. The status (**SUCCESSFUL/FAILED**) in the order execution result is not updated to the record of the order in the IoT platform database.

b. The order execution result notification sent by the IoT platform to the NA server does not contain **commandId**.

c. The query result of the NA server indicates that the final status of the order is **DELIVERED**.

NOTE

The preceding two message flows are used to explain the function of the **mid** field. Some message flows are simplified in the figures.

In scenarios where whether orders are sent to the device is of concern but the order execution is not, the device and codec do not need to implement the **mid** field.

If the **mid** field is not implemented, the NA server cannot obtain the order execution result from the IoT platform. Therefore, the NA server needs to implement the solution by itself. For example, after receiving the order execution result response (without **commandId**), the NA server can do as follows:

- Match the response with the order according to the sequence in which orders are delivered. In this way, when the IoT platform delivers multiple orders to the same device at the same time, the order execution result is matched with the delivered order incorrectly if packet loss occurs. Therefore, it is recommended that the NA server deliver only one order to the same device each time. After receiving the order execution result response, the NA server delivers the next order.
- The codec can add order-related information, such as an order code, to the **resultDetail** field of the order response to help identify the order. The NA server identifies the mapping between the order execution result response and the delivered order according to the information in the **resultDetail** field.

Do Not Use DirectMemory

The **DirectMemory** field directly calls the OS interface to apply for memory and is not controlled by the JVM. Improper use of the **DirectMemory** field may cause insufficient memory of the OS. Therefore, the DirectMemory cannot be used in codec plug-in code.

Example of improper use: Use **UNSAFE.allocateMemory** to apply for direct memory.

```
if ((maybeDirectBufferConstructor instanceof Constructor))
{
    address = UNSAFE.allocateMemory(1L);
    Constructor<?> directBufferConstructor;
    ...
}
else
{
    ...
}
```

Codec Input and Output Examples

The following table describes the definition of a service supported by a kind of water meter.

Service Type	Property Name	Property Description	Property Type (Data Type)
Battery	-	-	-

Service Type	Property Name	Property Description	Property Type (Data Type)
-	batteryLevel	Specifies the battery level in the unit of percent. The value ranges from 0 to 100.	int
Meter	-	-	-
-	signalStrength	Indicates the signal strength.	int
-	currentReading	Specifies the current read value.	int
-	dailyActivityTime	Specifies the daily activated communication duration.	string

The following shows the decode interface output for data reported by a device to the IoT platform.

```
{
   "identifier": "12345678",
   "msgType": "deviceReq",
   "data": [
      {
         "serviceId": "Meter",
         "serviceData": {
            "currentReading": "46.3",
            "signalStrength": 16,
            "dailyActivityTime": 5706
         },
         "eventTime": "20160503T121540Z"
      },
      {
         "serviceId": "Battery",
         "serviceData": {
            "batteryLevel": 10
         },
         "eventTime": "20160503T121540Z"
      }
   ]
}
```

The following shows the encode interface input when the IoT platform receives data reported by the device and sends a response to the device.

```
{
   "identifier": "123",
   "msgType": "cloudRsp",
   "request":[
      1,
      2
   ],
   "errcode": 0,
```



```
    "hasMore": 0
}
```

The following table describes the commands supported by a kind of water meter.

Basic Function	Category	Name	Command Parameter	Data Type	Enumerated Value
WaterMeter	Water meter	-	-	-	-
-	CMD	SET_TEMPERATURE_READ_PERIOD	-	-	-
-	-	-	value	int	-
-	RSP	SET_TEMPERATURE_READ_PERIOD_RSP	-	-	-
-	-	-	result	int	The value **0** indicates success. The value **1** indicates invalid input. The value **2** indicates execution failed.

The following shows the input parameters of the encode interface when the IoT platform sends an order to the device.

```
{
    "identifier": "12345678",
    "msgType": "cloudReq",
    "serviceId": "WaterMeter",
    "cmd": "SET_TEMPERATURE_READ_PERIOD",
    "paras": {
        "value": 4
    },
    "hasMore": 0
}
```

After the IoT platform receives a response from the device, the IoT platform invokes the decode interface for decoding. The decode interface output is as follows:

```
{
    "identifier": "123",
    "msgType": "deviceRsp",
    "errcode": 0,
    "body": {
```



```
        "result": 0
    }
}
```

Packaging the Codec

After the codec is developed, use the Maven to pack the codec into a JAR package and create it as a codec package.

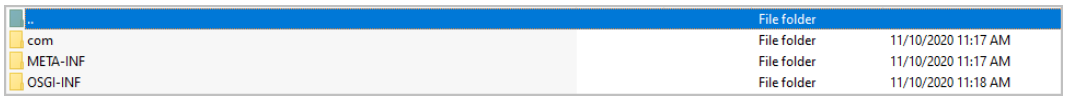
Maven Packaging

Step 1 Open the DOS window and access the directory where the **pom.xml** file is located.

Step 2 Run **mvn package**.

Step 3 After **BUILD SUCCESS** is displayed in the DOS window, open the **target** folder in the same directory as the **pom.xml** file to obtain the **.jar** package.

The naming rule of the **.jar** package is as follows: device type-manufacturer ID-device model-version.jar, for example: WaterMeter-Huawei-NBIoTDevice-version.jar.



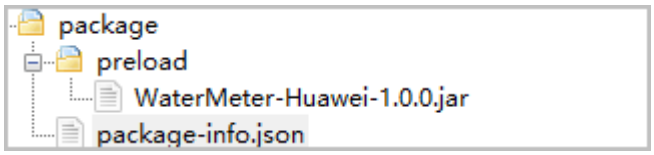
- The **com** directory stores **class** files.
- The **META-INF** directory stores description files of **.jar** packages under the OSGi framework, which are generated based on configurations in the **pom.xml** file.
- The **OSGI-INF** directory stores service configuration files and is used to register the codec as a service for the platform to call (only one .xml file can be called).
- Other **.jar** packages are **.jar** packages referenced by codecs.

----End

Preparing a Codec Package

Step 1 Create a folder named **package**, which contains the **preload/** sub-folder.

Step 2 Place the packaged **.jar** package in the **preload/** folder.



Step 3 In the **package** folder, create the **package-info.json** file. The fields and templates in this file are described as follows:

Note: The **package-info.json** file is encoded using UTF-8 without BOM. Only English characters are supported.

Table 3-4 Description of fields in the package-info.json file

Parameter	Description	Mandatory or Optional
specVersion	Specifies the version of the description file. The value is fixed at **1.0**.	Yes
fileName	Specifies the name of the software package. The value is fixed at **codec-demo**.	Yes
version	Specifies the version number of the software package. The version of the **package.zip** file must be the same as the value of **bundleVersion**.	Yes
deviceType	Specifies the device type, which must be the same as that defined in the profile.	Yes
manufacturerName	Specifies the manufacturer name, which must be the same as that defined in the profile. Otherwise, the **package-info.json** file cannot be uploaded to the IoT platform.	Yes
platform	Specifies the platform type, which is the operating system of the IoT platform on which the codec package runs. The value is fixed at **linux**.	Yes
packageType	Specifies the software package type. This field is used to describe the IoT platform module where the codec is deployed. The value is fixed at **CIGPlugin**.	Yes
date	Specifies the time when a packet is sent. The format is as follows: yyyy-MM-dd HH-mm-ss. For example, 2017-05-06 20:48:59.	No
description	Specifies the self-defined description about the software package.	No
ignoreList	Specifies the list of bundles to be ignored. The default value is **null**.	Yes
bundles	Specifies the description of a bundle. **Note**: A bundle is a **.jar** package in a compressed package. Only one bundle needs to be described.	Yes

Table 3-5 Description of the bundles field

Parameter	Description	Mandatory or Optional
bundleName	Specifies the bundle name, which is consistent with the value of **Bundle-SymbolicName** in the **pom.xml** file.	Yes
bundleVersion	Specifies the bundle version, which must be the same as the value of **version**.	Yes
priority	Specifies the bundle priority. This parameter can be set to the default value **5**.	Yes
fileName	Specifies the codec file name.	Yes
bundleDesc	Describes the bundle function.	Yes
versionDesc	Describes the functions and features of different versions.	Yes

Template of the **package-info.json** file

```
{
   "specVersion":"1.0",
   "fileName":"codec-demo",
   "version":"1.0.0",
   "deviceType":"WaterMeter",
   "manufacturerName":"Huawei",
   "description":"codec",
   "platform":"linux",
   "packageType":"CIGPlugin",
   "date":"2017-02-06 12:16:59",
   "ignoreList":[],
   "bundles":[
   {
      "bundleName": "WaterMeter-Huawei",
      "bundleVersion": "1.0.0",
      "priority":5,
      "fileName": "WaterMeter-Huawei-1.0.0.jar",
      "bundleDesc":"",
      "versionDesc":""
   }]
}
```

Step 4 Select all files in the **package** folder and compress them into a **package.zip** file.

Note: The **package.zip** file cannot contain the **package** directory.

----End

3.4.5 Downloading and Uploading a Codec

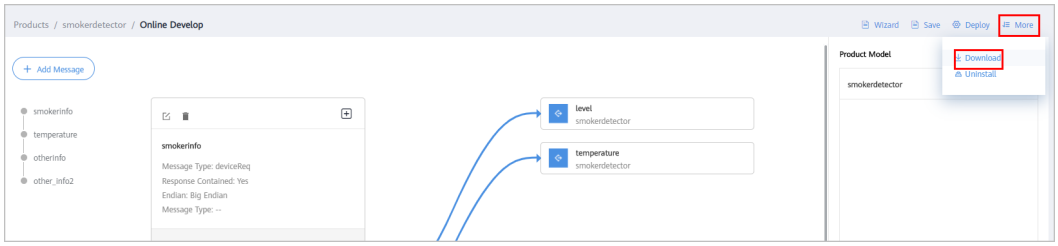
The codec developed online can be download to a local directory. The local codec can also be uploaded to any other IoT platform.

Downloading a Codec

Step 1 Log in to the **IoTDA** 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 Choose **Codec Development** > **Online Develop**. On the page displayed, select **More** in the upper right corner and choose **Download** to download the codec package.



----End

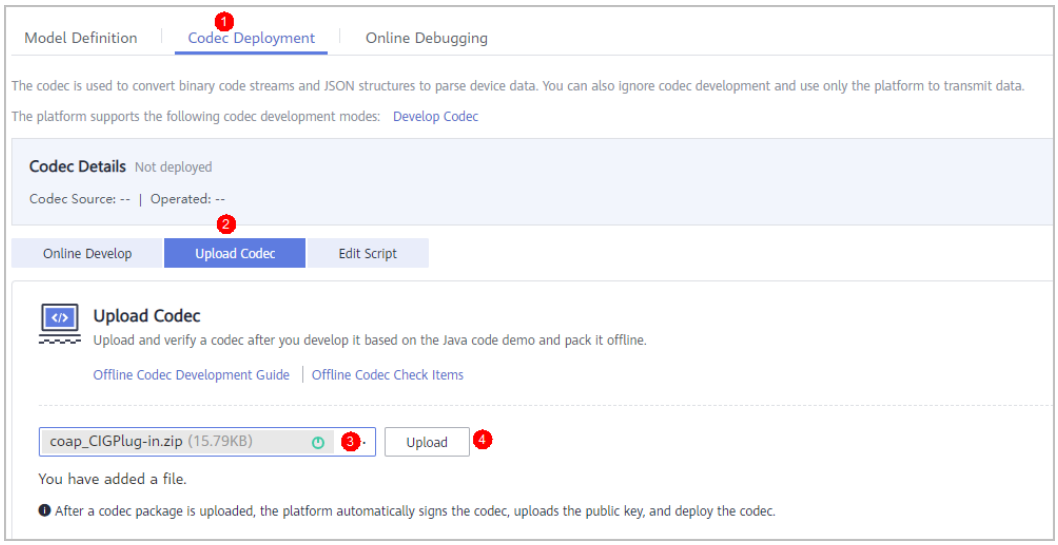
Uploading a Codec

If a codec (such as a codec developed offline) is available on the local host, the codec can be uploaded to the IoT platform.

Step 1 Log in to the **IoTDA** 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 product details page, click **Codec Development**, select **Upload Codec**, select a local codec package, and click **Upload**.



NOTE

Device Type, **Model**, and **Manufacturer ID** of the codec package must be the same as those of the product.

If the message "Offline codec uploaded successfully" is displayed, the codec has been deployed on the IoT platform.

----End

3.5 Online Debugging

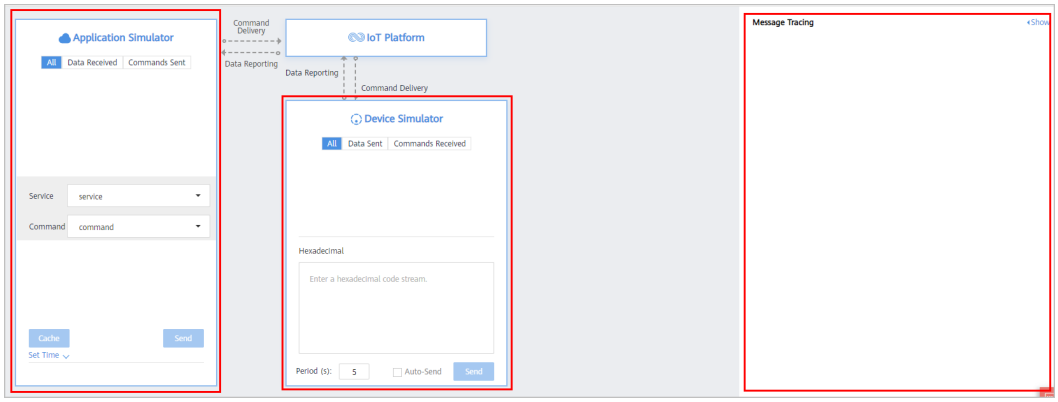
Overview

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.

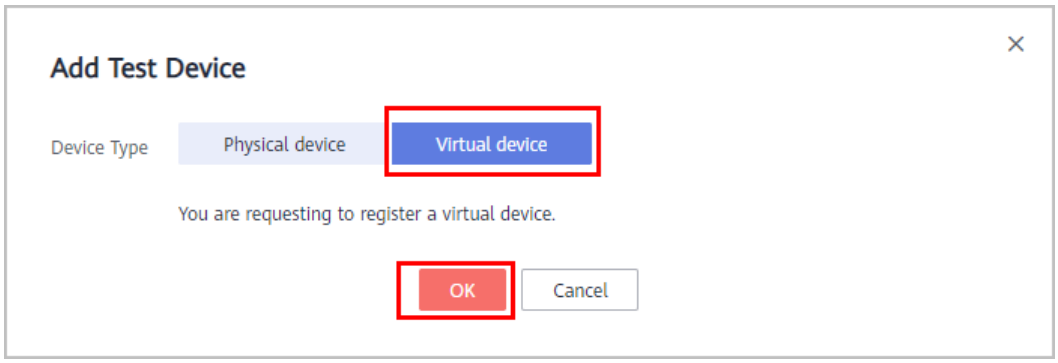
Commissioning a Product by Using a Virtual Device

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:

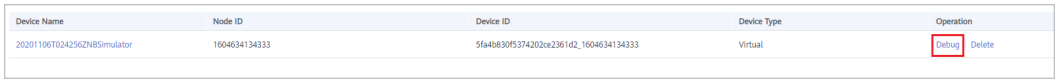


Step 1 On the product details page, click the **Online Debugging** tab and click **Add Test Device**.

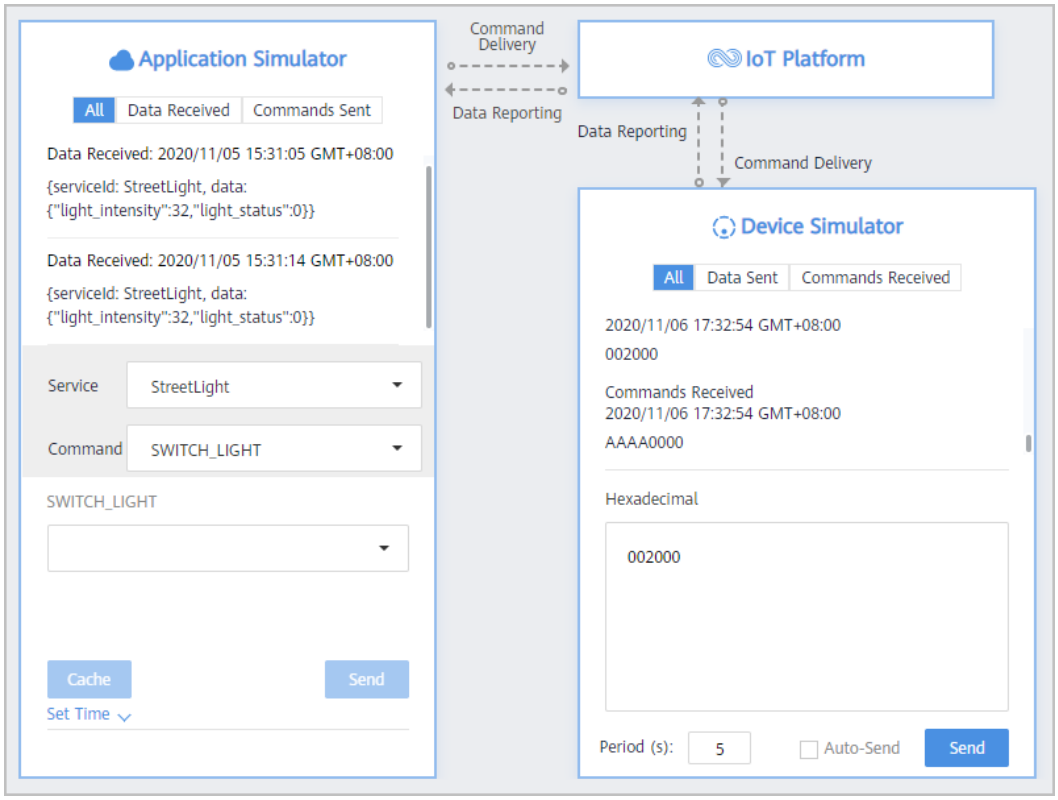
Step 2 In the **Add Test Device** dialog box, select **Virtual device** for **Device Type** and click **OK**. The virtual device name contains **Simulator**. Only one virtual device can be created for each product.



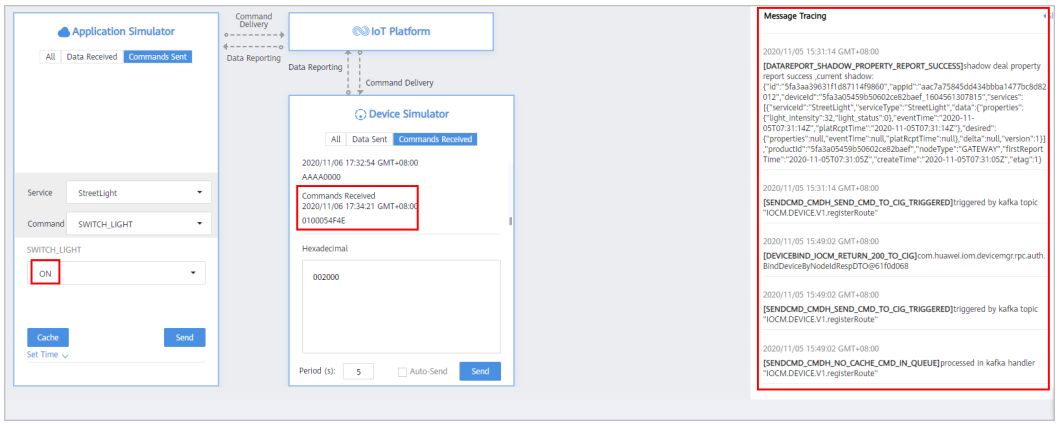
Step 3 In the device list, select the new virtual device and click **Debug** to enter the **Online Debugging** page.



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**.



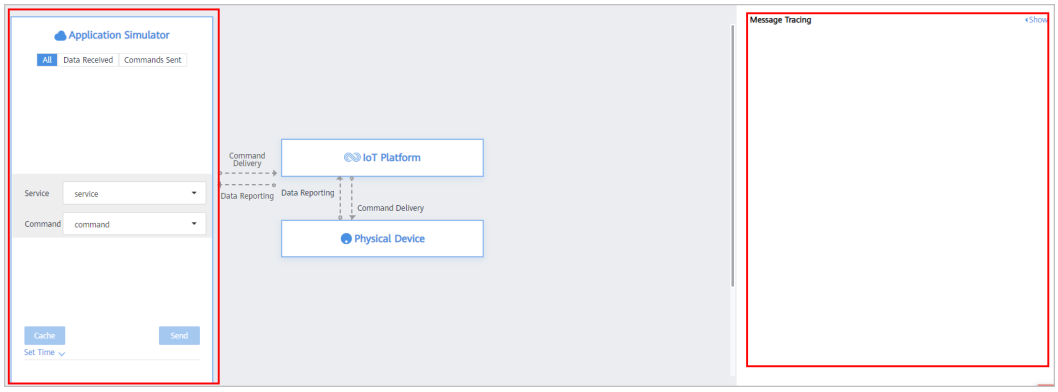
Step 5 Deliver a command in **Application Simulator**. View the received command (for example, a hexadecimal code stream) in **Device Simulator** and the processing logs of the IoT platform in **Message Tracing**.



----End

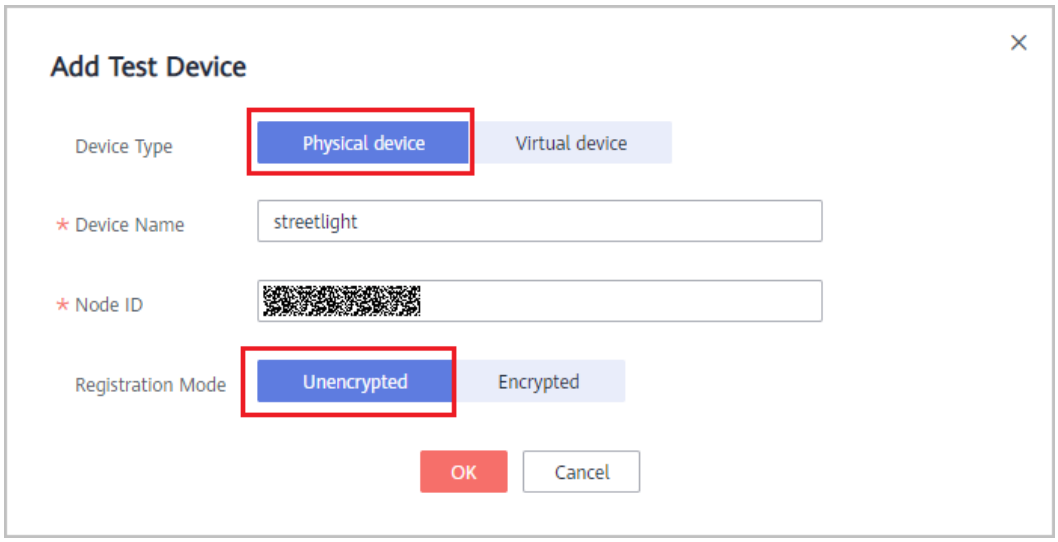
Debugging a Product by Using a Physical Device

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, select **Online Debugging** 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 **Connection Authentication**. After the device is connected to the platform, perform the debugging.

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

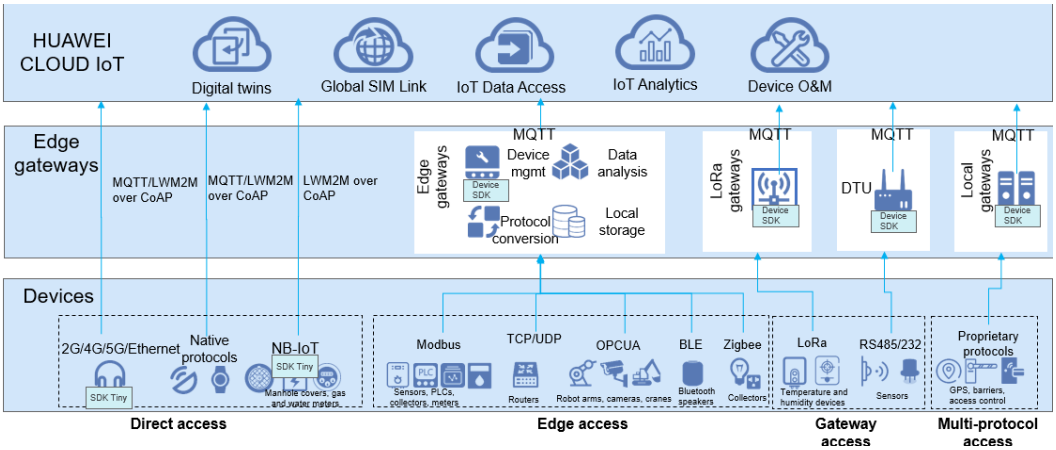
Step 4 Simulate a scenario where a control command is remotely delivered. In **Application Simulator**, Set **Service** to **StreetLight**, **Command** to **SWITCH_LIGHT**, and **Command Value** to **ON**, and click **Send**. The street lamp is turned on.

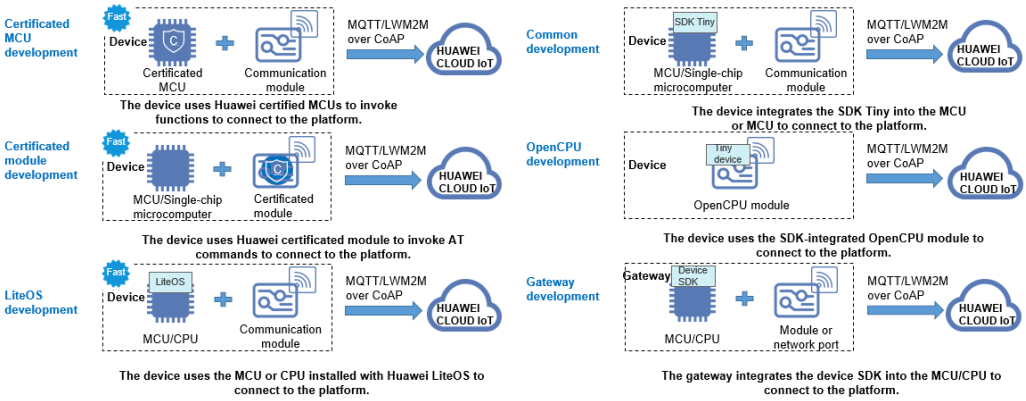
----End

4 Development on the Device Side

4.1 Device Access Guide

The HUAWEI CLOUD IoT platform 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.





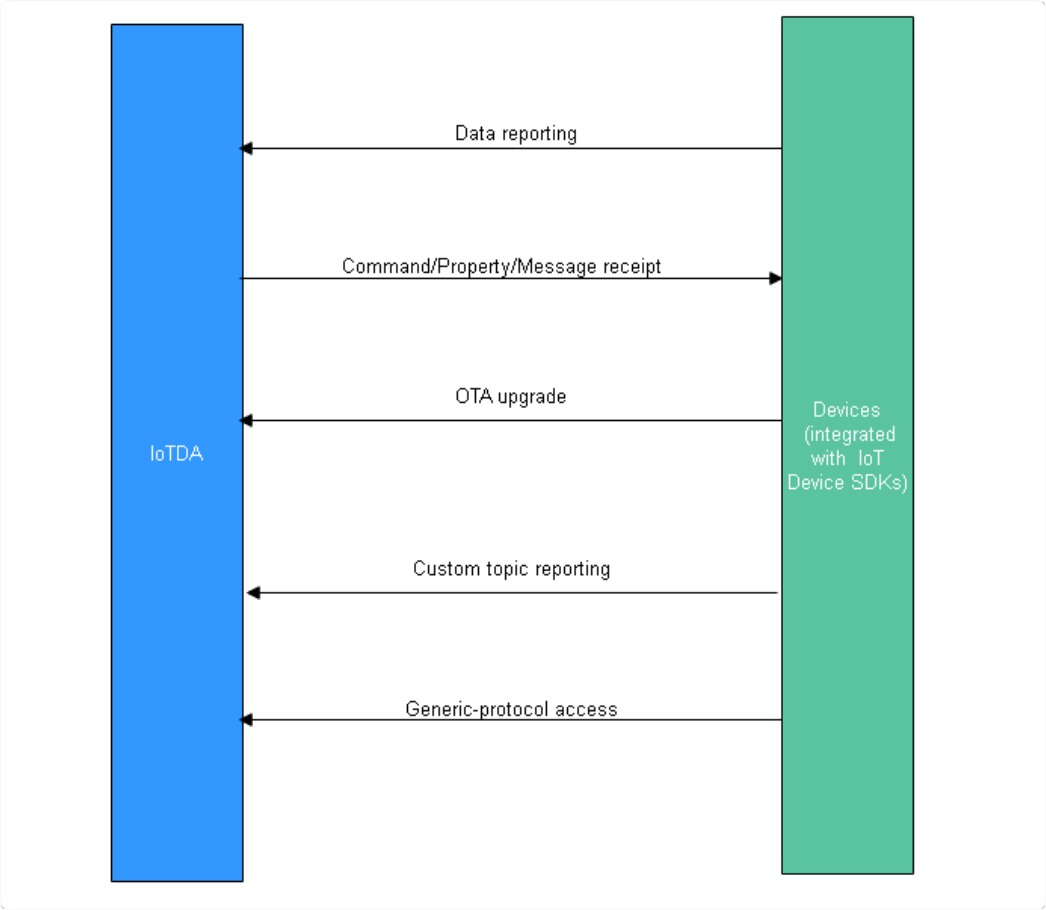
Development Mode	Feature	Scenario	Difficulty Level
Certificated MCU development	The IoT Device SDK Tiny has been pre-integrated into the main control unit (MCU) and can call methods to connect to the platform.	Devices need to be quickly put into commercial use, with low R&D costs. Devices are connected 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. For details, see **4.4 Using Huawei-Certified Modules for Access**.	★
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.	★★ ★★

Development Mode	Feature	Scenario	Difficulty Level
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.	★★★

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.



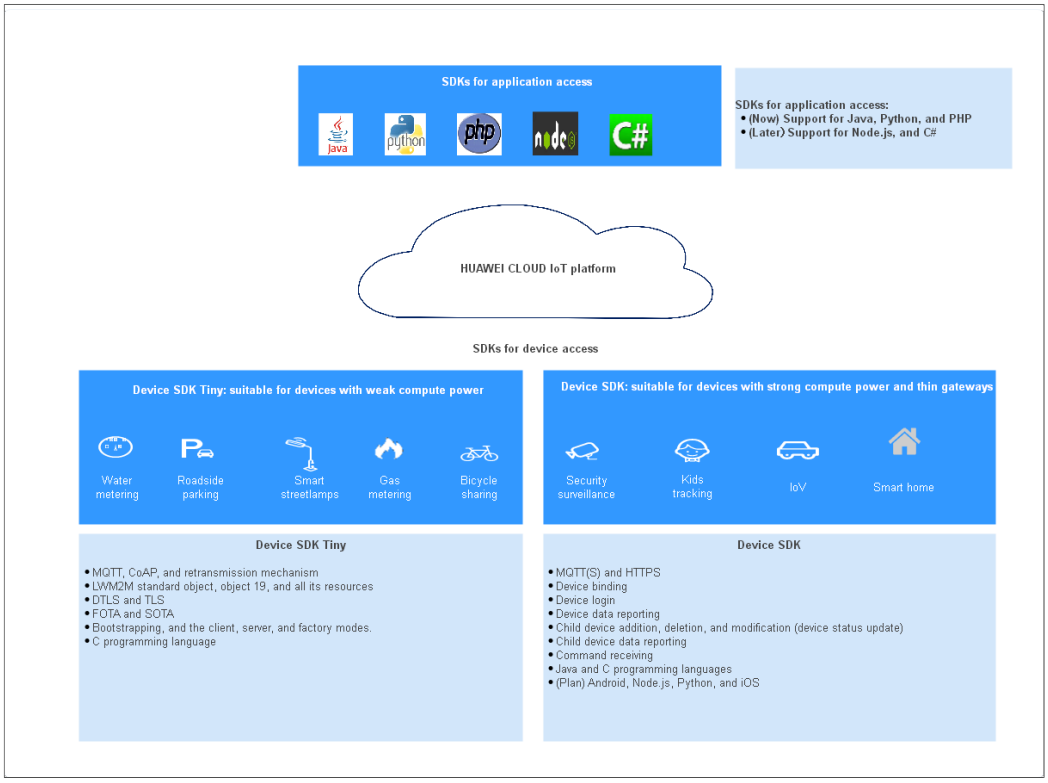
1. Create a product on the IoTDA console or by calling the API **Creating a Product**.
2. Register the device on the IoTDA console or by calling the API **Registering a Device**.
3. Implement the functions demonstrated in the preceding figure, including reporting messages/properties, receiving commands/properties/messages, OTA upgrades, topic customization, and generic-protocol access (see **Demo**).

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
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), and Android	C, Java, C#, and Android
IoT Device SDK Tiny	> 32 KB	> 128 KB	> 100 MHZ	No special requirements	C



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

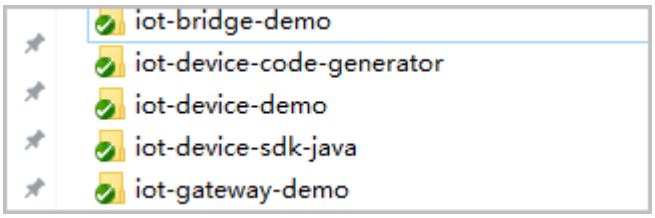
- **IoT Device SDK (C)**
- **IoT Device SDK (Java)**
- **IoT Device SDK (C#)**
- **IoT Device SDK (Android)**
- **IoT Device SDK Tiny**

4.2.2 IoT Device SDK (Java)

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 of common directly connected devices

iot-gateway-demo: demo code of gateways

iot-bridge-demo: demo code of the bridge, which demonstrates how to bridge a TCP device to the platform

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 compile and install the SDK.

Creating a Product

We provide a smokeDetector product model to facilitate understanding. The smoke detector can report the smoke density, temperature, humidity, and smoke alarms, and execute the ring alarm command. The following procedures use the smoke detector as an example to experience functions such as message reporting and property reporting.

Step 1 Log in to the **IoTDA** console to view the MQTTS device access domain name, and save the address.

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

Step 3 Set the parameters as prompted and click **Create Now**.

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 value can contain letters, numbers, underscores (_), and hyphens (-).
Protocol	Select **MQTT**.
Data Type	Select **JSON**.
Manufacturer	Customize the manufacturer name. The value can contain letters, numbers, underscores (_), and hyphens (-).
Define Product Model	

Product Model	In this example, we import a product model, rather than using a preset product model. For details, see **Uploading a Product Model**.
Industry	Select the industry to which the product model belongs.
Device Type	Customize the device type.

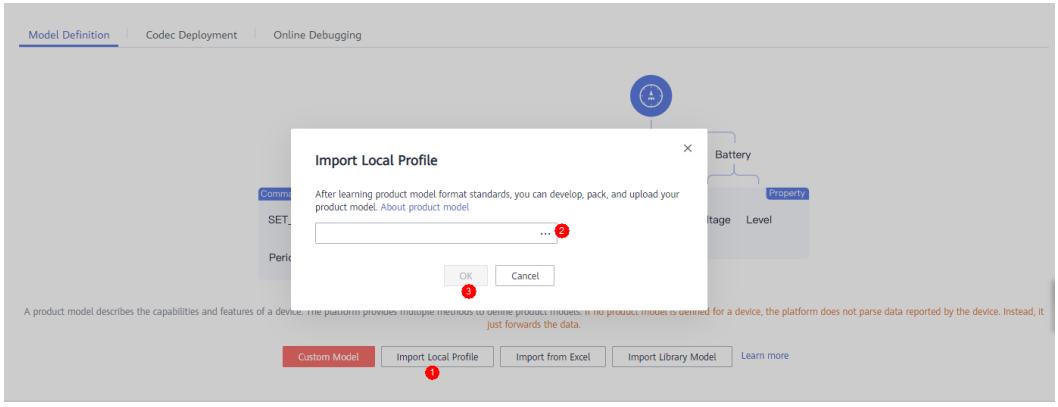
----End

Uploading a Product Model

Step 1 Download the product model smokeDetector to obtain the product model file.

Step 2 Select the product created in **3** and click **View** to access its details.

Step 3 On the **Model Definition** tab page, click **Import Local Profile** to upload the product model file obtained in **1**.



----End

Registering a Device

Step 1 Choose **Devices** > **All Devices**, and click **Individual Register** in the upper right corner.

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.

Parameter	Description
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 the Device

1. Enter the device ID and secret obtained in **Registering a Device** and the device interconnection information obtained in **1** in the format of ***ssl://Domain name:Port***.

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

2. Establish a connection. Call the API **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:

```
2019-12-26 11:02:02  INFO MqttConnection:88 - Mqtt client connected. address :ssl://iot-acc.cn-north-4.myhuaweicloud.com:8883
```

3. After the device is created and connected, it can be used for communication. You can call the API **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 the API **getClient** of the IoT Device SDK to obtain the client from the device.
2. Call the API **reportDeviceMessage** to enable the client to report a device message. In the message sample below, messages are reported periodically.

```
while (true) {


    device.getClient().reportDeviceMessage(new DeviceMessage("hello"), new ActionListener() {
     @Override
     public void onSuccess(Object context) {
     }

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

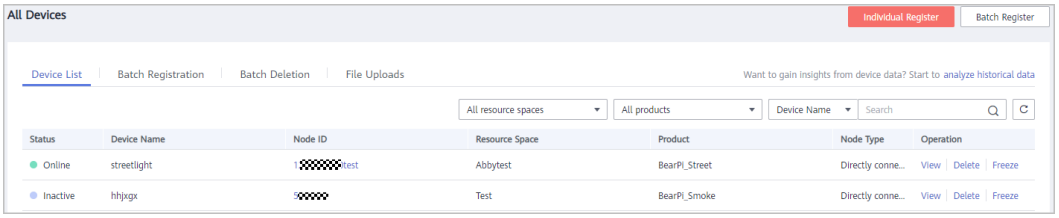


```
        });

        Thread.sleep(10000);
    }
```

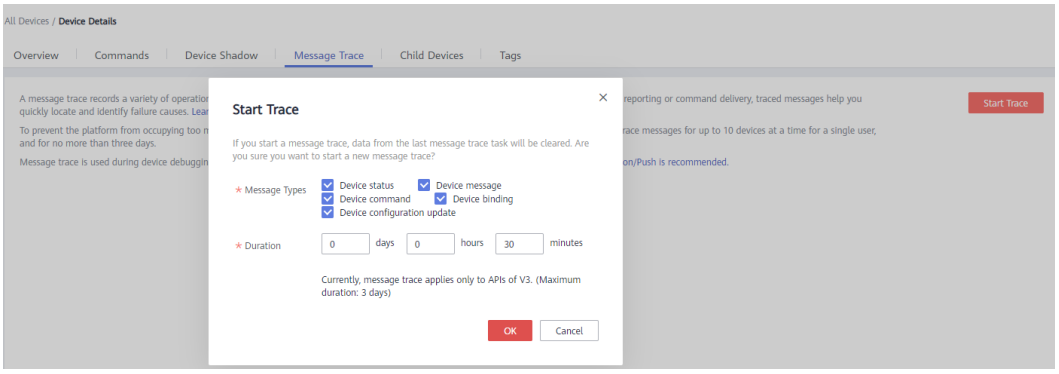
3. Replace the device parameters with the actual values in the **main** function of the **MessageSample** class, and run this class. Then view the logs about successful connection and message reporting.

```
2019-12-26 11:02:02  INFO MqttConnection:88 - Mqtt client connected. address :ssl://iot-acc.cn-north-4.myhuaweicloud.com:8883
2019-12-26 11:02:02  INFO MqttConnection:214 - publish message topic =  $oc/devices/test_testDevice/sys/messages/up, msg = {"name":null,"id":null,"content":"hello","object_device_id":null}
```

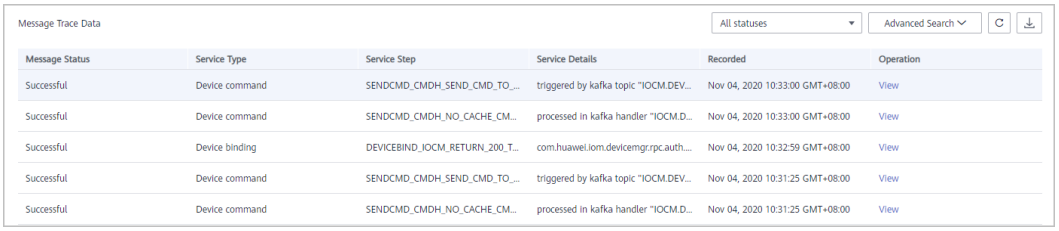
4. On the IoTDA console, choose **Devices** > **All Devices** and check whether the device is online.



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



6. View the messages received by the platform.



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", alarm);
        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.

```
"C:\Program Files (x86)\Java\jdk1.8.0_73\bin\java.exe" ...
2019-12-28 10:39:07  INFO MqttConnection:140 - try to connect to ssl://iot-acc.cn-north-4.myhuaweicloud.com:8883
2019-12-28 10:39:07  INFO MqttConnection:147 - connect success ssl://iot-acc.cn-north-4.myhuaweicloud.com:8883
2019-12-28 10:39:07  INFO MqttConnection:87 - Mqtt client connected. address :ssl://iot-acc.cn-north-4.myhuaweicloud.com:8883
2019-12-28 10:39:08  INFO MqttConnection:213 - publish message topic =  $oc/devices/5e06bfee334dd4f33759f5b3_demo/sys/properties/report
2019-12-28 10:39:08  INFO PropertySample:90 - pubMessage success
```

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

Reading and Writing Properties

Call the **setPropertyListener** method of the client to set the property callback API. 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){
```

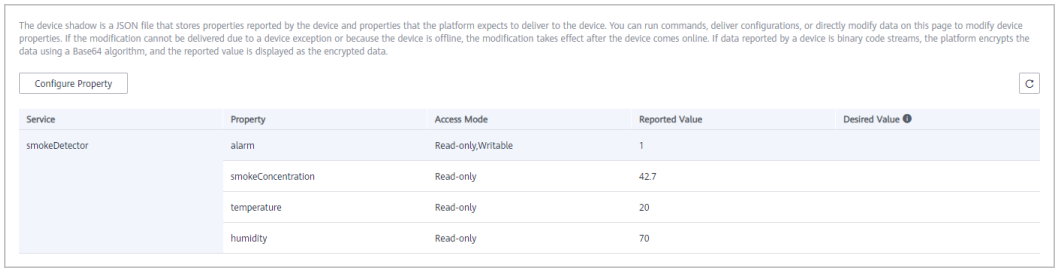


```
            log.info("OnPropertiesSet, serviceId =  " + serviceProperty.getServiceId());

            // Traverse properties.
            for (String name :serviceProperty.getProperties().keySet()){
                log.info("property name = "+ name);
                log.info("set property value = "+ serviceProperty.getProperties().get(name));
                if (name.equals("alarm")){
                    // Change the local value.
                    alarm = (Integer) serviceProperty.getProperties().get(name);
                }
            }

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

    // Process property reading.
    @Override
    public void onPropertiesGet(String requestId, String serviceId) {

        log.info("OnPropertiesGet " + serviceId);
        Map<String ,Object> json = new HashMap<>();
        Random rand = new Random();
        json.put("alarm", alarm);
        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");

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

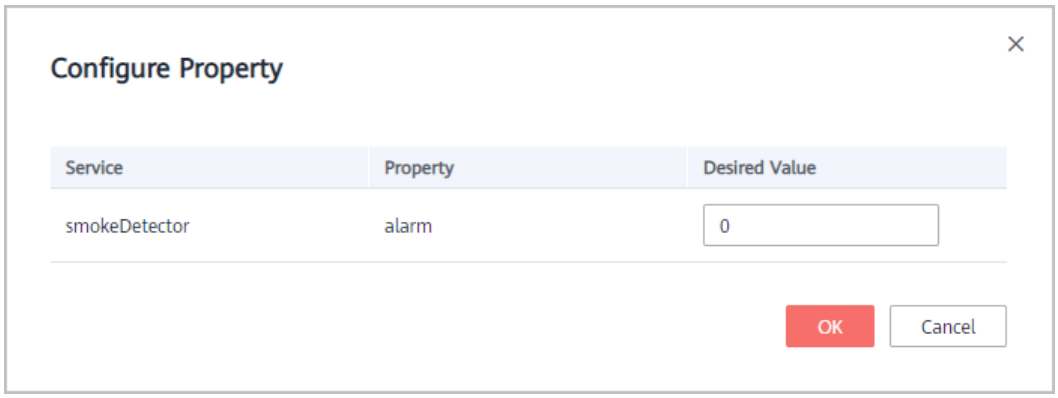
Note:

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



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



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

```
2019-12-28 14:16:27  INFO MqttConnection:66 - messageArrived topic =  $oc/devices/5e06bfee334dd4f33759f5b3_demo/sys/proper
2019-12-28 14:16:27  INFO PropertySample:53 - OnPropertiesSet, serviceId =  smokeDetector
2019-12-28 14:16:27  INFO PropertySample:57 - property name = alarm
2019-12-28 14:16:27  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 API 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.
            client.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.

```
2019-12-28 15:03:36  INFO MqttConnection:66 - messageArrived topic =  $oc/devices/test_testDevice/sys/commands/request_id=4, msg = {"paras":{"duration":20},"service_id":"smo
2019-12-28 15:03:36  INFO CommandSample:62 - onCommand, serviceId = smokeDetector
2019-12-28 15:03:36  INFO CommandSample:63 - onCommand , name = ringAlarm
2019-12-28 15:03:36  INFO CommandSample:64 - onCommand, paras =  {duration=20}
2019-12-28 15:03:36  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.

We use the smokeDetector example 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**.

```
// 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** APIs.
   - 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 APIs 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 the humidity field, because it is read-only.
    }

    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 the set API 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 the set API 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.

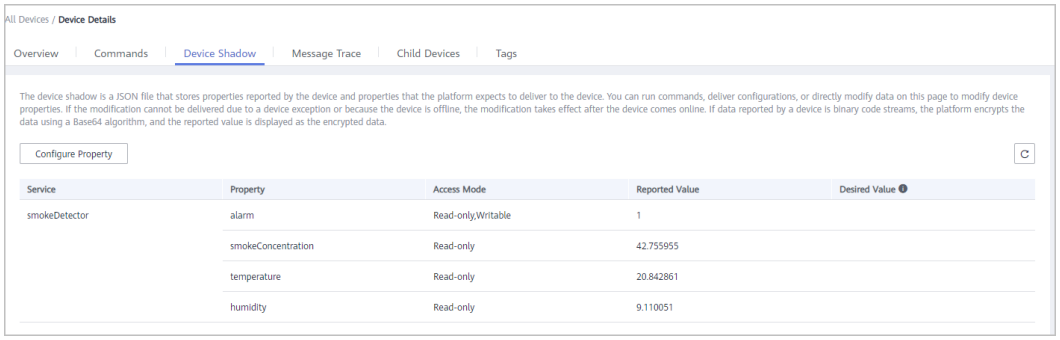
```
// Enable periodic property reporting.
smokeDetectorService.enableAutoReport(10000);
```

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

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

```
2019-12-28 15:26:26  INFO MqttConnection:140 - try to connect to ssl://iot-acc.cn-north-4.myhuaweicloud.com:8883
2019-12-28 15:26:26  INFO MqttConnection:147 - connect success ssl://iot-acc.cn-north-4.myhuaweicloud.com:8883
2019-12-28 15:26:26  INFO MqttConnection:87 - Mqtt client connected. address :ssl://iot-acc.cn-north-4.myhuaweicl
connect ok
2019-12-28 15:26:26  INFO MqttConnection:213 - publish message topic =  $oc/devices/5e06bfee334dd4f33759f5b3_demo
```

View the device shadow on the platform.



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

```
2019-12-28 15:44:29  INFO MqttConnection:66 - messageArrived topic =  $oc/devices/test_testDevice/sys/properties/set/request_id=2, msg = {"services":[{"pr
2019-12-28 15:44:29  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.

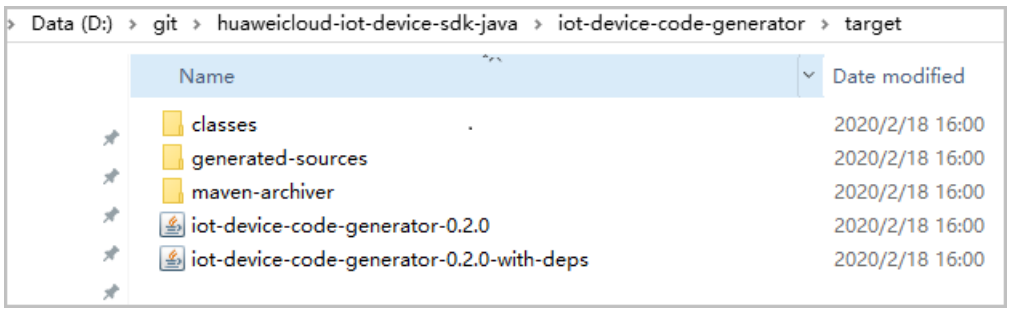
```
2019-12-28 15:44:29  INFO MqttConnection:66 - messageArrived topic =  $oc/devices/test_testDevice/sys/commands/request_id=1, msg = {"paras":{"duration":20},
2019-12-28 15:44:29  INFO DeviceServiceSample$SmokeDetectorService:53 - ringAlarm  duration = 20
2019-12-28 15:44:29  INFO MqttConnection:213 - publish message topic =  $oc/devices/test_testDevice/sys/commands/response/request_id=1, msg = {"paras":null,
```

Using the Code Generator

The SDK provides a device 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.
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_cb097d20d77b4240adf1f33d36b3c278_smokeDetector.zip** file to disk D.
4. Go to the **iot-device-code-generator\target** directory and run the **java -jar iot-device-code-generator-0.2.0-with-deps.jar D:\smokeDetector_cb097d20d77b4240adf1f33d36b3c278_smokeDetector.zip** command.

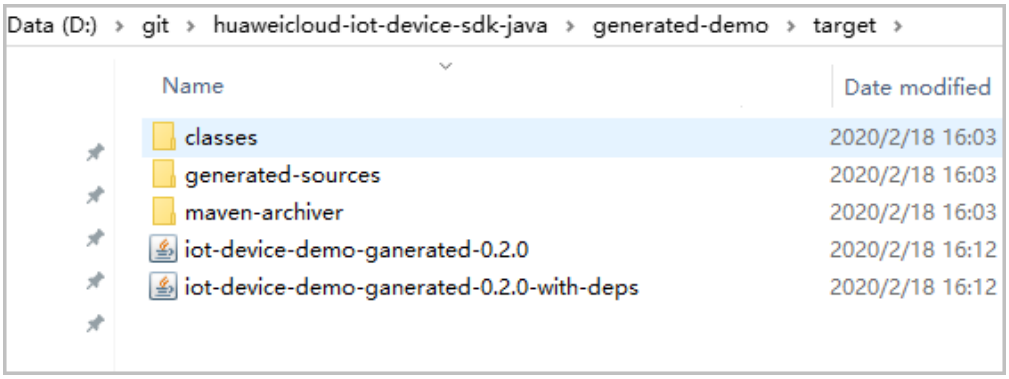
```
D:\git\huaweicloud-iot-device-sdk-java> java -jar iot-device-code-generator\target\iot-device-code-generator-0.2.0-with-deps.jar D:\smokeDetector_cb097d20d77b4240adf1f33d36b3c278_smokeDetector.zip
2020-02-18 16:10:18  INFO DeviceCodeGenerator:117 -          File path: D:\git\huaweicloud-iot-device-sdk-java\generated-demo\src\main\java\com\huaweicloud\sdk\iot\device\demo\smokeDetectorService.java
2020-02-18 16:10:18  INFO DeviceCodeGenerator:45 - 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.



2. Run the **java -jar target\iot-device-demo-ganerated-0.2.0-with-deps.jar 5e06bfee334dd4f33759f5b3_demo ***** command.

   You need to specify the device ID and password in the command to run the generated demo.

```
D:\git\huaweicloud-iot-device-sdk-java\generated-demo>java -jar target\iot-device-demo-ganerated-0.2.0-with-deps.jar 5e06bfee334dd4f33759f5b3_demo ******
2020-02-18 16:17:19  INFO IoTDevice:59 - create device: 5e06bfee334dd4f33759f5b3_demo
2020-02-18 16:17:20  INFO MqttConnection:147 - try to connect to ssl://iot-acc.cn-north-4.myhuaweicloud.com:8883
```

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 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: The **onAddSubDevices** API of **AbstractGateway** can store child device information. Additional processing is not required, and the **onAddSubDevices** API 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, the **onDeleteSubDevices** API is overridden to add the link release logic, and the parent class **qit onDeleteSubDevices** 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
property 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 the **channelRead0** API 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 more information 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.

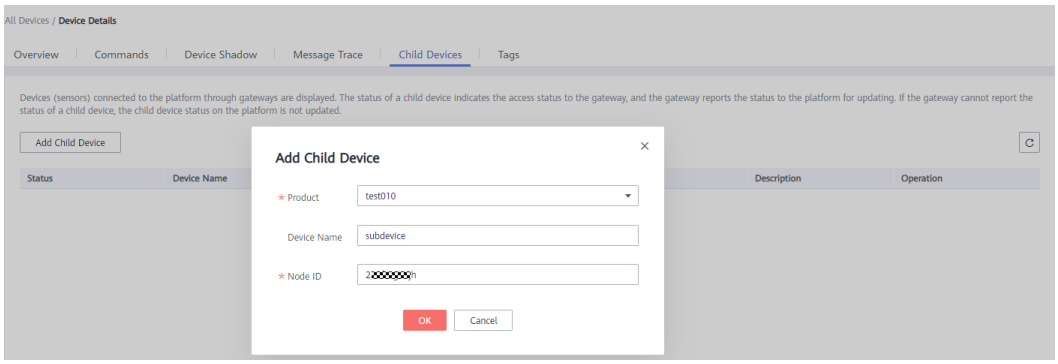
- **Using iot-gateway-demo**

  a. Register a gateway with the platform.

  b. 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");
```

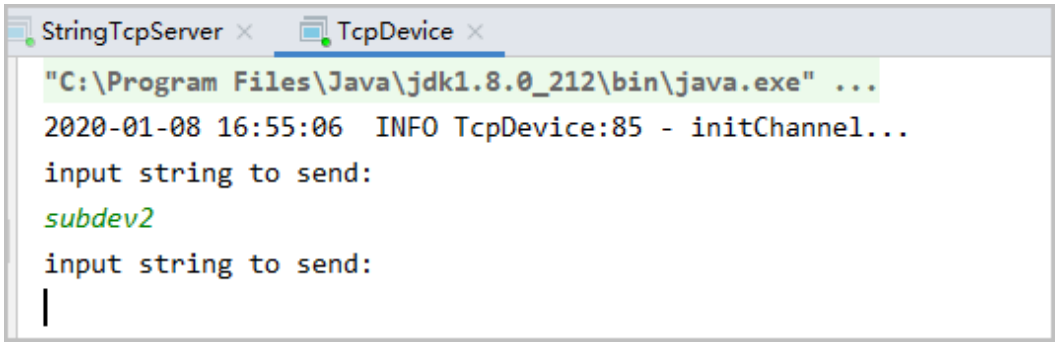
  c. After the gateway is displayed as **Online** on the platform, add a child device.



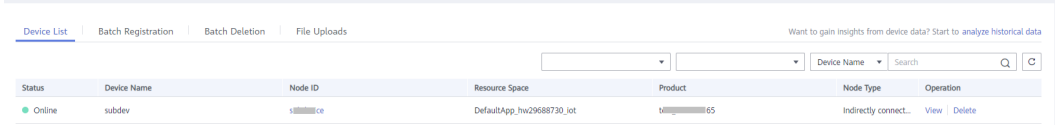
  A log similar to the following is displayed on the gateway:

2020-02-11 09:42:17 INFO SubDevicesFilePersistence:75 - add subdev: ffff

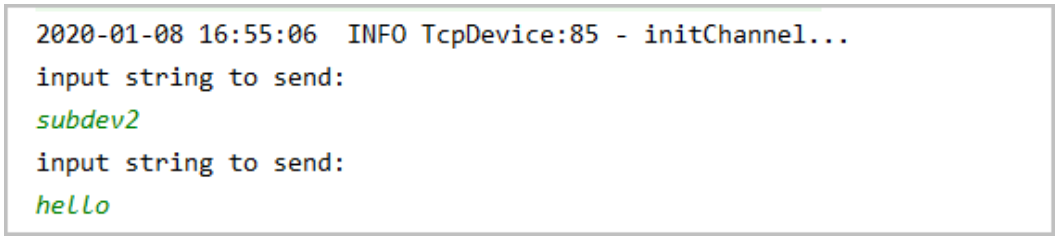
d. Run the **TcpDevice** class. After the connection is established, enter the nodeId of the child device.



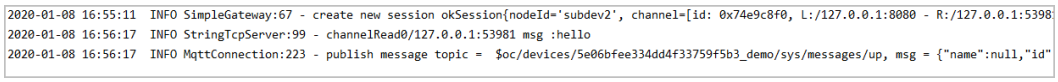
e. Check whether the child device is online on the platform.



f. Enable the child device to report messages.

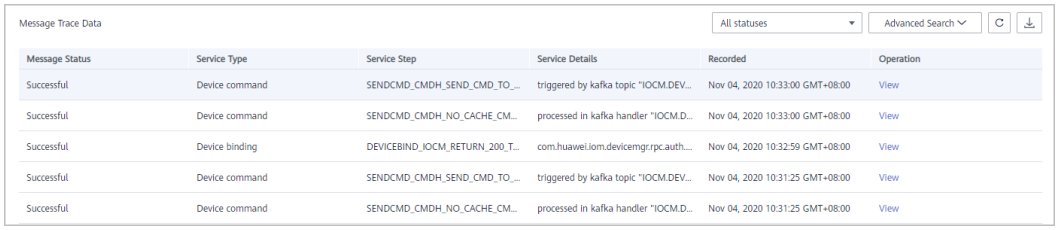


Logs similar to the following show that messages are reported.



g. 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.



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**.

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**.

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**.

4.2.6 IoT Device SDK Tiny (C)

The IoT Device SDK Tiny is lightweight interconnection middleware suitable for devices that have WAN capabilities, low power consumption, and limited storage and computing resources. You only need to call APIs to enable these devices to access the platform, report data, and receive commands. For details, see **Huawei LiteOS SDK Development Guide**.

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.

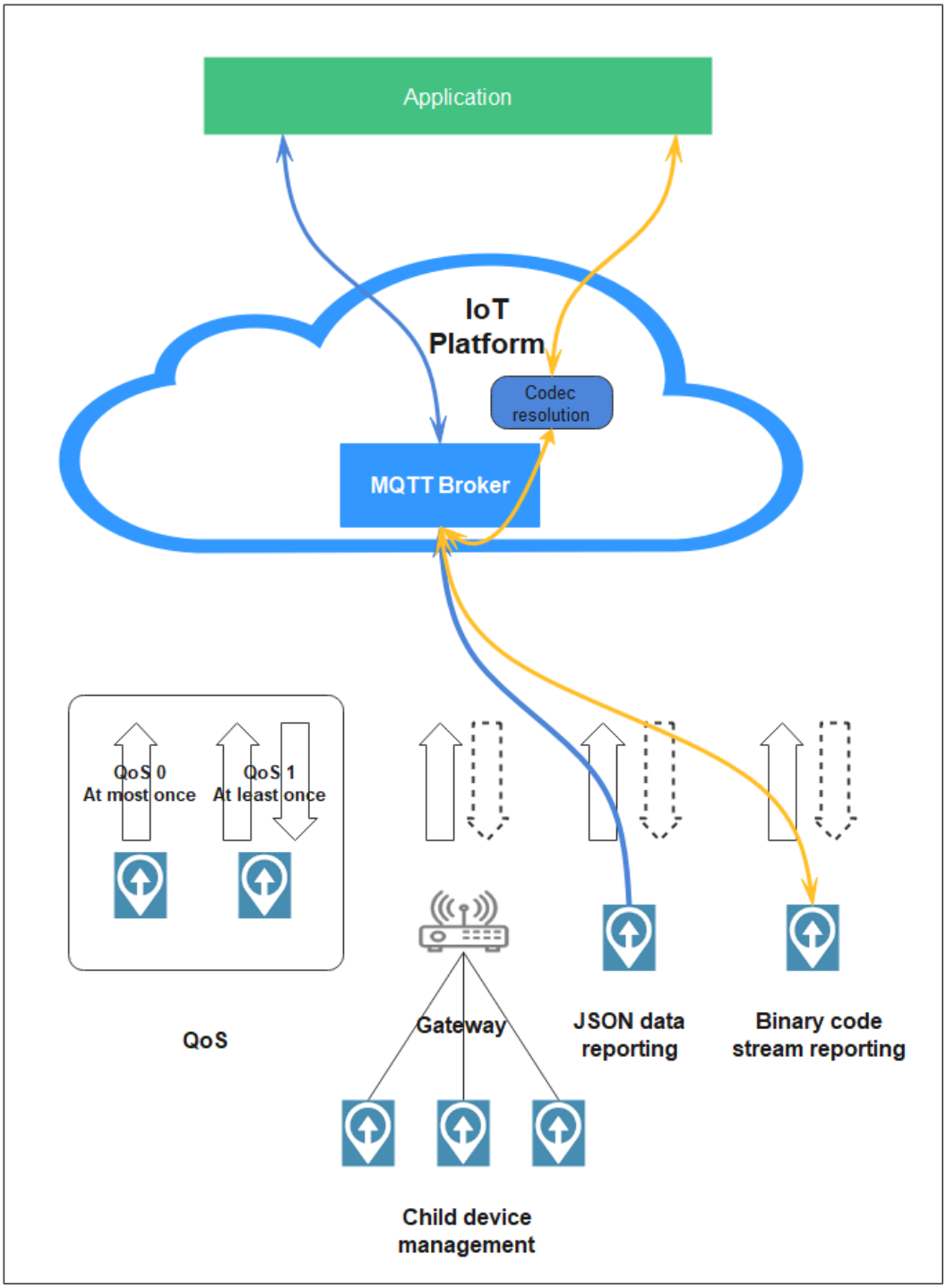
4.3 Using MQTT Demos for Access

4.3.1 MQTT

Introduction

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**.

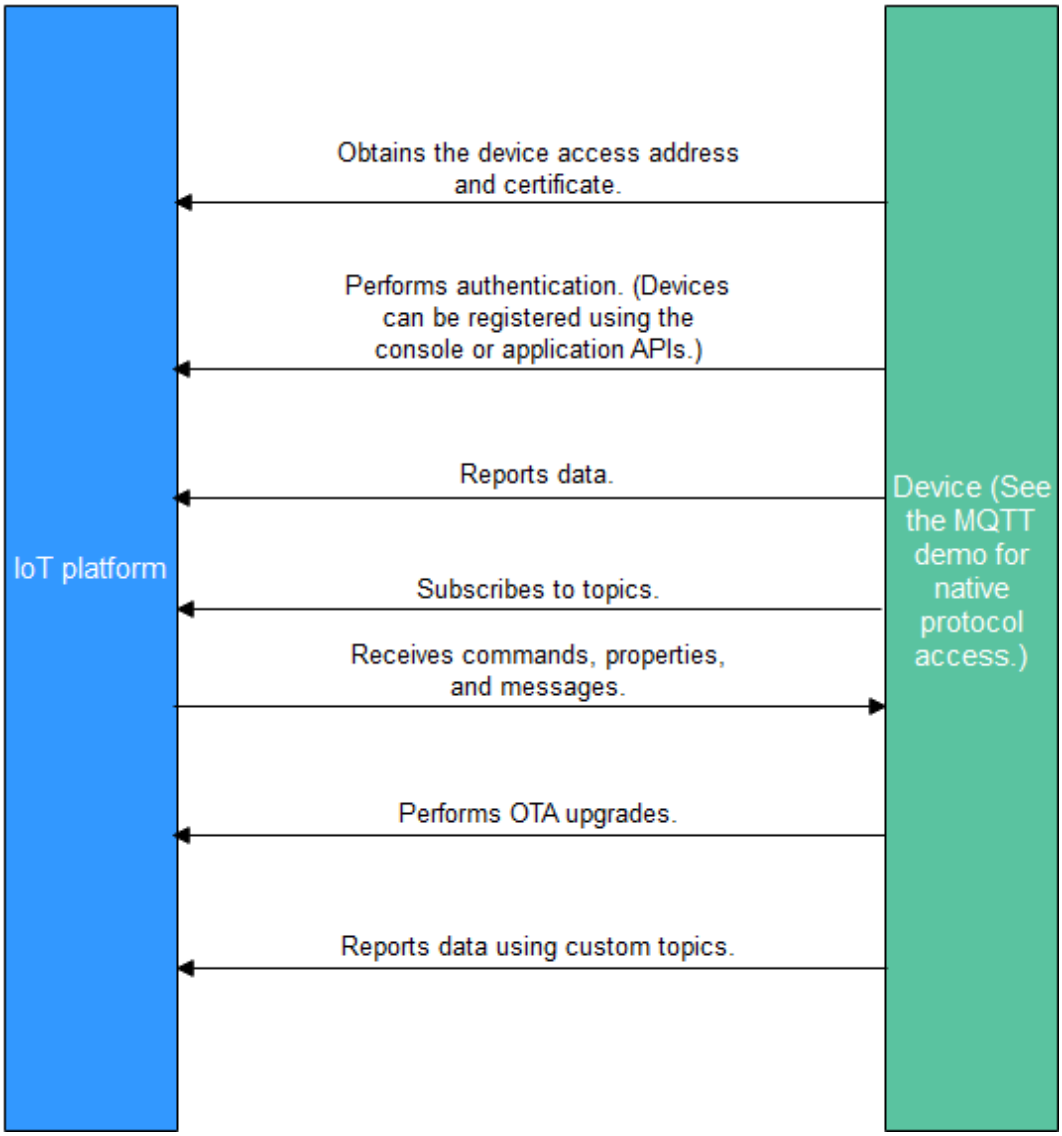
MQTTS is a variant of MQTT that uses TLS encryption. MQTTS devices communicate with the platform using encrypted data transmission.



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 by using the IoTDA console or calling the API **Creating a Product**.
2. Register a device by using 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 **Connecting MQTT Devices**.

Constraints

Item	Constraint
Supported MQTT version	3.1.1
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 v1, TLS v1.1, and TLS v1.2)
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
Topic customization	Not supported
Message publishing and subscription	A device can only publish and subscribe to messages of its own topics.
Maximum number of subscriptions per subscription request	No limit

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	**Reporting a Device Message**	$oc/devices/{device_id}/sys/messages/up	Device	Platform

Category	Function	Topic	Publisher	Subscriber
related topics	**Delivering a Device Message**	$oc/devices/{device_id}/sys/messages/down	Platform	Device
Device command related topics	**Delivering a Device Command**	$oc/devices/{device_id}/sys/commands/request_id={request_id}	Platform	Device
	Returning a Command Response	$oc/devices/{device_id}/sys/commands/response/request_id={request_id}	Device	Platform
Device property related topics	**Reporting Device Property Data**	$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
	Returning a Response to a Property Query (The response will not affect device properties or shadows.)	$oc/devices/{device_id}/sys/properties/get/response/request_id={request_id}	Device	Platform

Category	Function	Topic	Publisher	Subscriber
	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 Topics**.

TLS Support for MQTT

TLS is recommended for secure transmission between devices and the platform. Currently, TLS V1.0, V1.1, and V1.2 are supported. TLS V1.0 and V1.1 will soon be deprecated. Therefore, TLS V1.2 is 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 MQTT.fx

This section uses MQTT.fx as an example to describe how to connect devices to the platform using the native MQTT protocol. MQTT.fx is a widely used MQTT client that makes it easy to verify whether devices can interact with the platform to publish or subscribe to messages.

Prerequisites

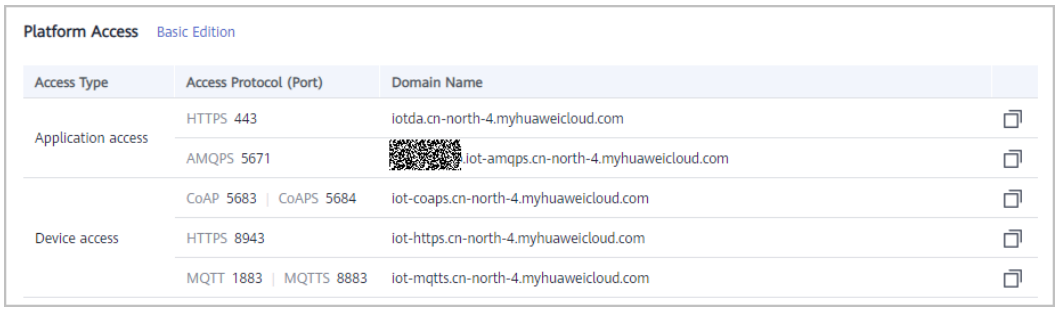
- You have **registered** a HUAWEI CLOUD account.
- You have completed **real-name authentication** on HUAWEI CLOUD.
- You have subscribed to the IoTDA service.

Obtaining Device Access Information

Perform the following procedure to obtain device access information on the IoTDA console:

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

Step 2 Click **Overview** in the navigation pane, view the device access information, and record the domain names and ports.



NOTE

For devices that cannot be connected to the platform using a domain name, run the **ping** *Domain name* command in the CLI to obtain the corresponding IP address. Then you can connect the devices to the platform using the IP address. The IP address is variable and needs to be set using a configuration item.

----End

Creating a Product and Registering a Device

Step 1 (Optional) Create a product that uses MQTT. If an MQTT product already exists, skip this step.

1. Choose **Products** in the navigation pane and click **Create Product** in the upper right corner.
2. Set the parameters as prompted and click **Create**.

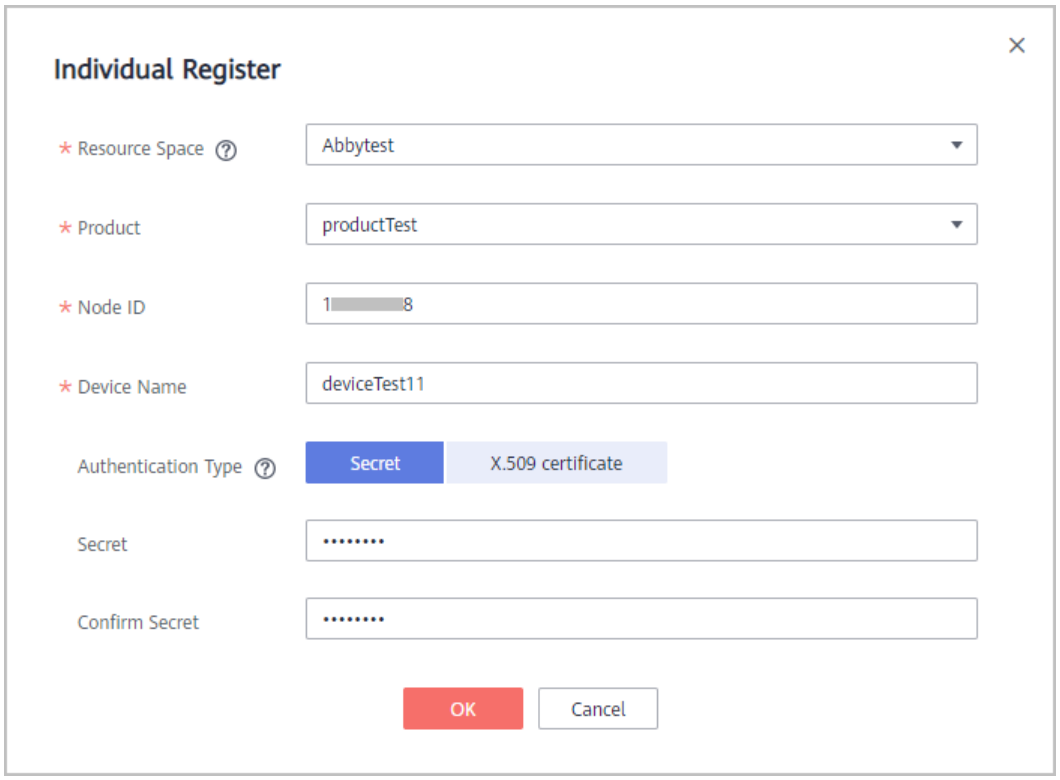
Set Basic Info	
Resource Space	The platform automatically allocates the created product to the default resource space. If you want to allocate the product to another resource space, select the resource space from the drop-down list. If a **resource space** does not exist, create it first.
Product Name	Customize the product name. The value can contain letters, numbers, underscores (_), and hyphens (-).

Protocol	Select **MQTT**.
Data Type	Select **JSON**.
Manufacturer	Customize the manufacturer name. The value can contain letters, numbers, underscores (_), and hyphens (-).
Define Product Model	
Product Model	You are advised to use a product model preset on the platform to experience device access. **This section uses WaterMeter as an example. You can also select other product models.**
Industry	Select the industry to which the product model belongs.
Device Type	If a product model preset on the platform is used, the device type is automatically matched and does not need to be manually specified.

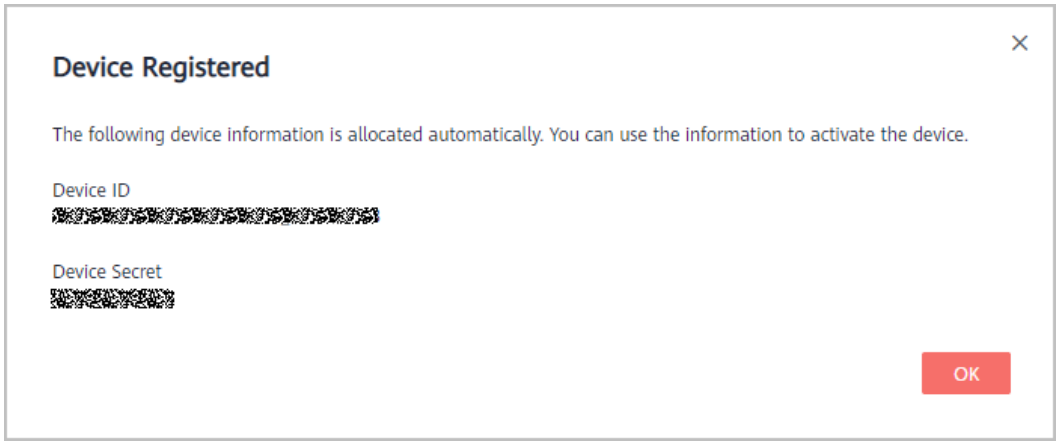
Step 2 Register a device.

1. Choose **Devices** > **All Devices**, and click **Individual Register** in the upper right corner.
2. Set the parameters as prompted and click **OK**.

Parameter	Description
Resource Space	Ensure that the device and the product created in **1** belong to the same resource space.
Product	Select the product created in **1**.
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 secret used for device access. If the secret is left blank, the platform automatically generates one.



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



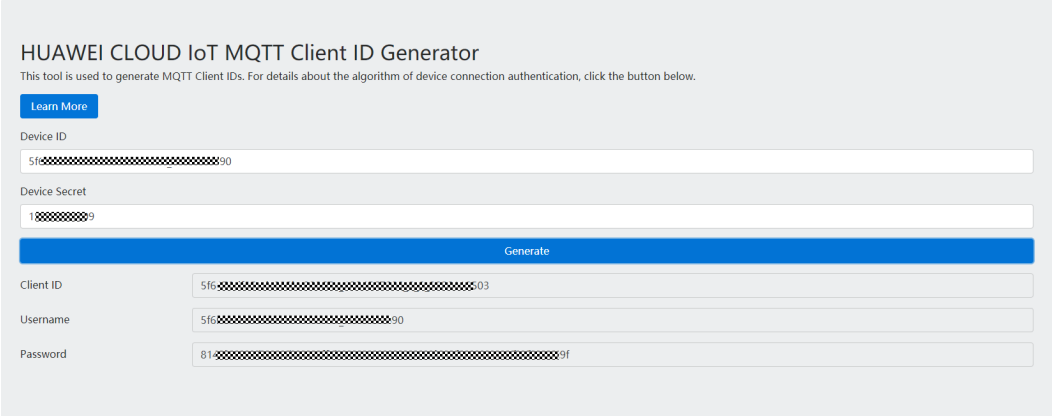
----End

Performing Connection Authentication

You can use the MQTT.fx tool to connect devices to the platform by referring to **Device Connection Authentication** in the *API Reference*.

Step 1 Visit the **MQTT.fx website** and download and install the latest version of MQTT.fx.

Step 2 Go to the **IoTDA client ID generator page**, enter the device ID and secret generated after **registering a device** to generate connection information (including **ClientId**, **Username**, and **Password**).



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

Each device performs authentication using the MQTT CONNECT message, which must contain all information of the **clientId**. After receiving a CONNECT message,

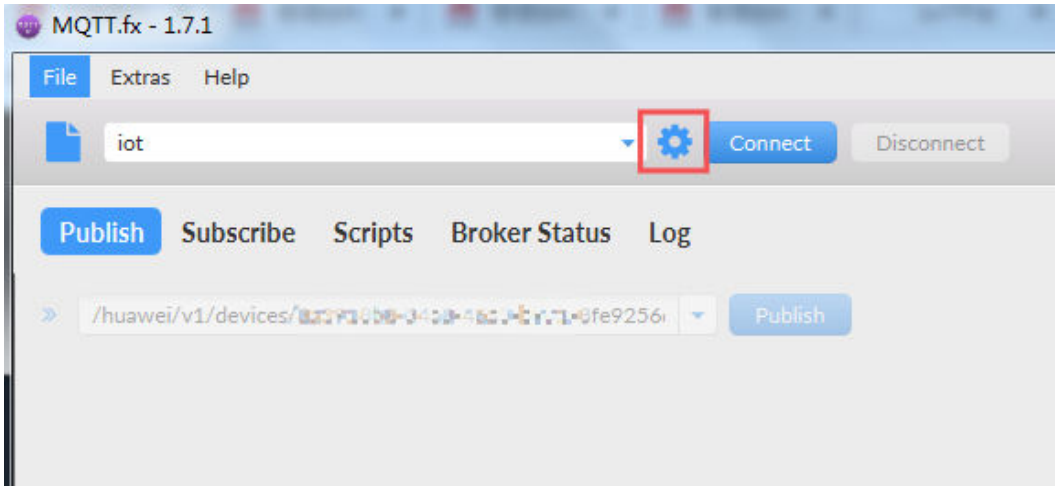
the platform checks the authentication type and password digest algorithm of the device.

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

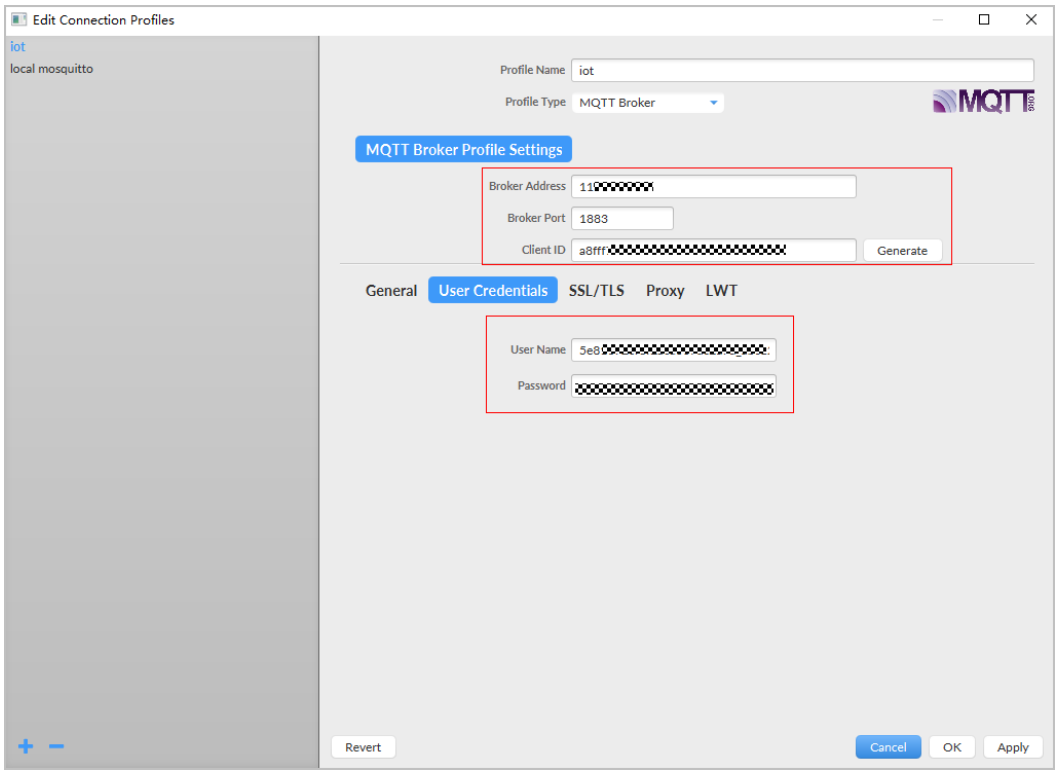
- If the timestamp is verified using the HMAC-SHA256 algorithm, the platform checks whether the message timestamp is consistent with the platform time and then checks whether the password is correct.
- If the timestamp is not verified using the HMAC-SHA256 algorithm, the timestamp must also be contained in the CONNECT message, but the platform does not check whether the time is correct. In this case, only the password is checked.

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

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

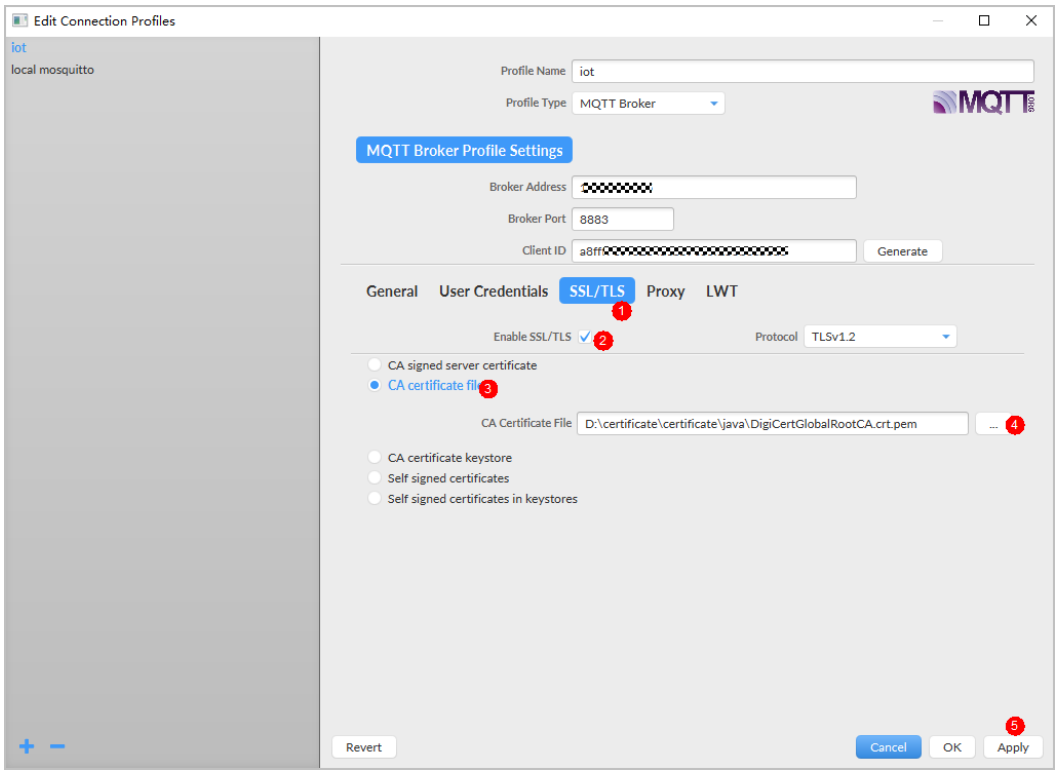


Step 4 Configure authentication parameters and click **Apply**.

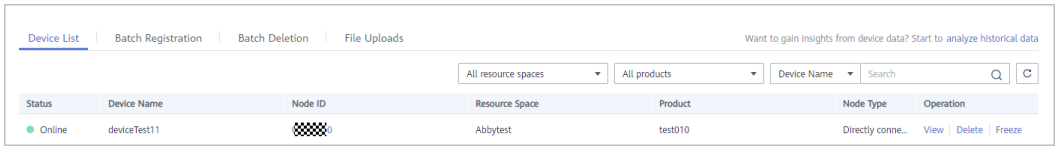


Parameter	Description
Broker Address	Enter the **device connection address** (domain name) obtained from the IoTDA console. If the device cannot be connected using a domain name, enter the IP address obtained in **2**.
Broker Port	The default value is **1883**.
Client ID	Device ClientId obtained in **2**.
User Name	DeviceId obtained in **2**.
Password	Encrypted device secret obtained in **2**.

If you choose secure access, set **Broker Port** to **8883**, download the **certificate**, and load the Java certificate in .pem format.



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



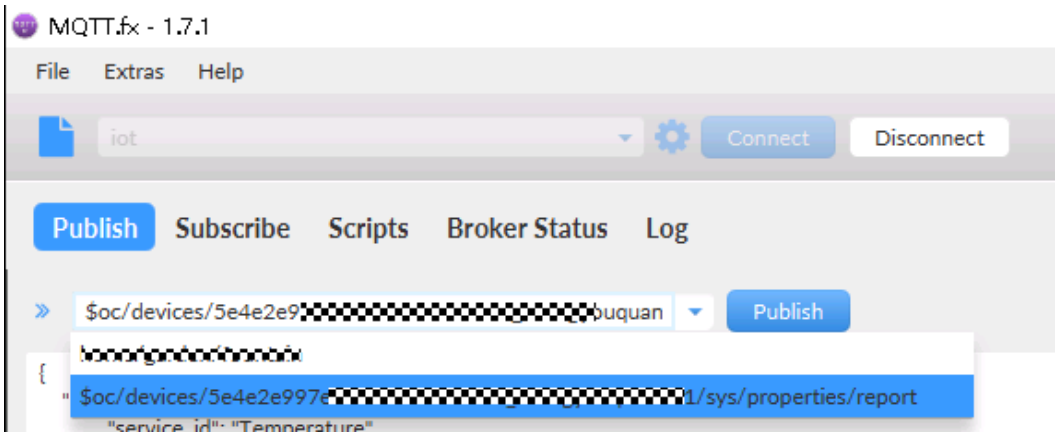
----End

Reporting Data

You can use the MQTT.fx tool to report data to the platform by referring to **Reporting Device Properties** in the *API Reference*.

If the device reports data through the MQTT channel, the data needs to be sent to a specific topic in the format **$oc/devices/{device_id}/sys/properties/report**. For devices that each has a different secret, specify **device_id** as the device ID returned upon successful device registration.

Step 1 Enter the API address in the format of "$oc/devices/*{device_id}*/sys/properties/report", for example, **$oc/devices/5e4e2e92ac-164aefa8fouquan1/sys/properties/report**.



Step 2 Enter the data to report.

Request parameters

Field	Mandatory	Type	Description
services	Yes	List<ServiceProperty>	Service data list. (For details, see the ServiceProperty structure in the following table.)

ServiceProperty structure

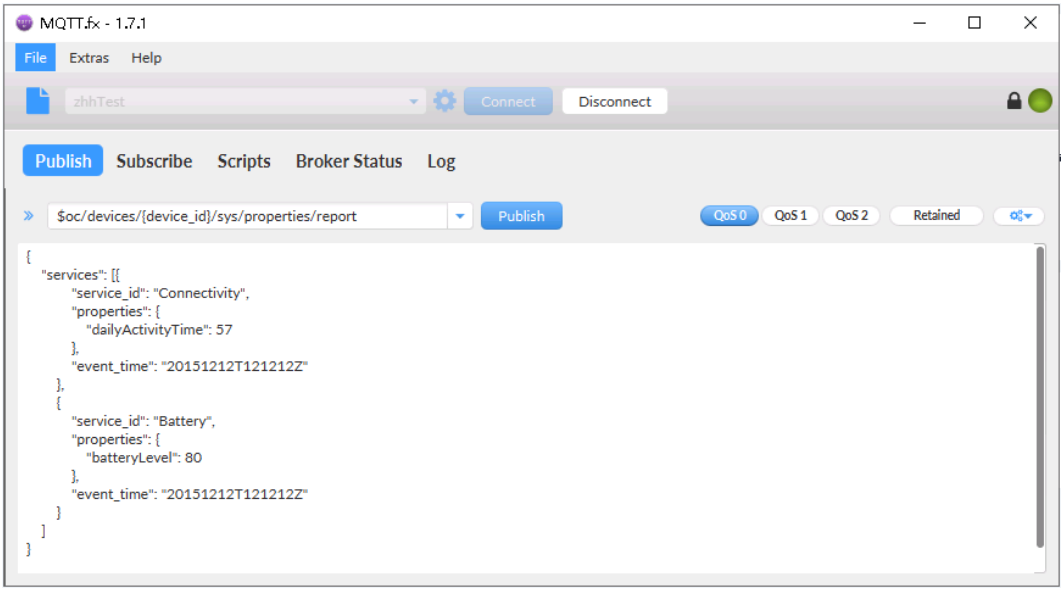
Field	Mandatory	Type	Description
service_id	Yes	String	Service ID.
properties	Yes	Object	Service properties, which are defined in the product model associated with the device.
eventTime	No	String	Indicates the UTC time when the device collects data. The format is yyyyMMddTHHmmssZ, for example, **20161219T114920Z**.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

Example request

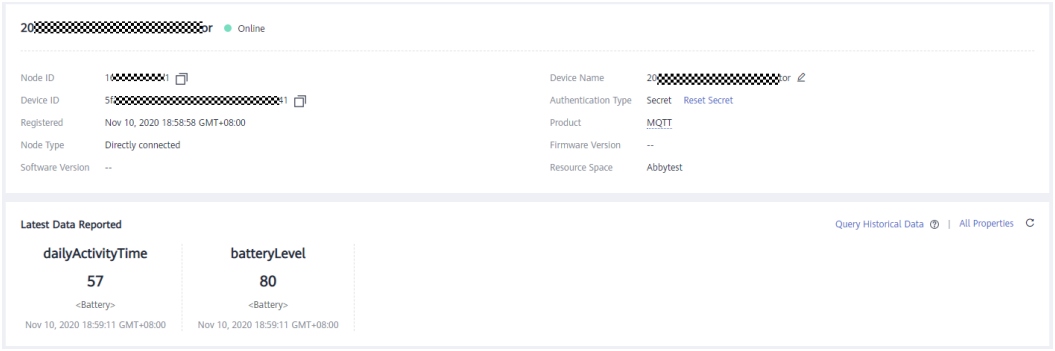
```
{
   "services": [{
          "service_id": "Connectivity",
          "properties": {
             "dailyActivityTime": 57
```



```
        },
        "event_time": "20151212T121212Z"
    },
    {
        "service_id": "Battery",
        "properties": {
            "batteryLevel": 80
        },
        "event_time": "20151212T121212Z"
    }
  ]
}
```



Step 3 Click **Publish**. Then you can check whether the device successfully reports data on the platform.



----End

Advanced Experience

After using MQTT.fx to connect a simulated MQTT device to the platform, you may understand how the MQTT device interacts with the platform through open APIs over MQTTS.

To better experience the IoTDA service, develop real-world applications and devices and connect them to the platform. For details, see **IoTDA Developer Guide**.

4.3.3 Java Demo

Introduction

This section uses Java as an example to describe how to connect devices to the platform over MQTTS/MQTT and how to **report data** and **deliver commands** using **platform APIs**. For details about device access in other languages, see **Obtaining Resources**.

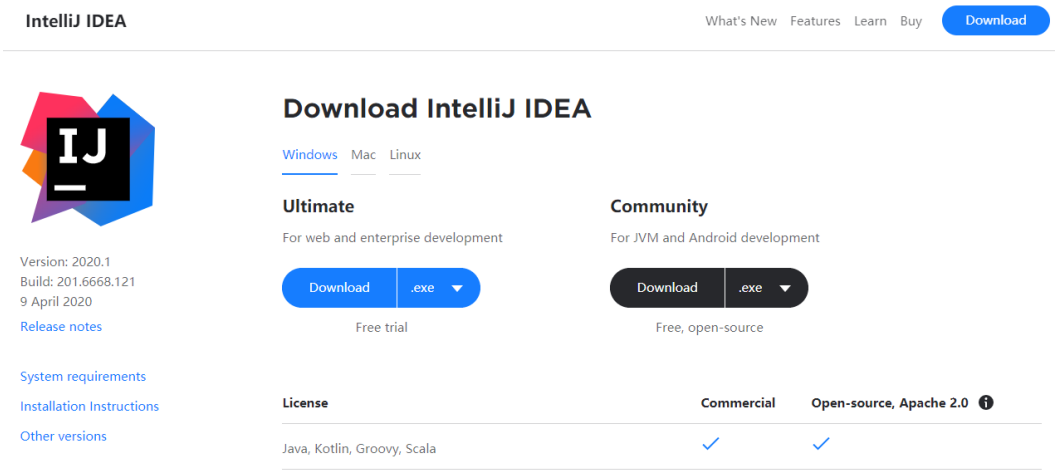
Prerequisites

- You have installed IntelliJ IDEA by following the instructions provided in **Installing IntelliJ IDEA**.
- You have obtained the device access address from the **IoTDA console**. For details, see **Platform Connection Information**.
- You have created a product and device on the **IoTDA console**. For details, see **Creating a Product**, **Registering an Individual Device**, or **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.

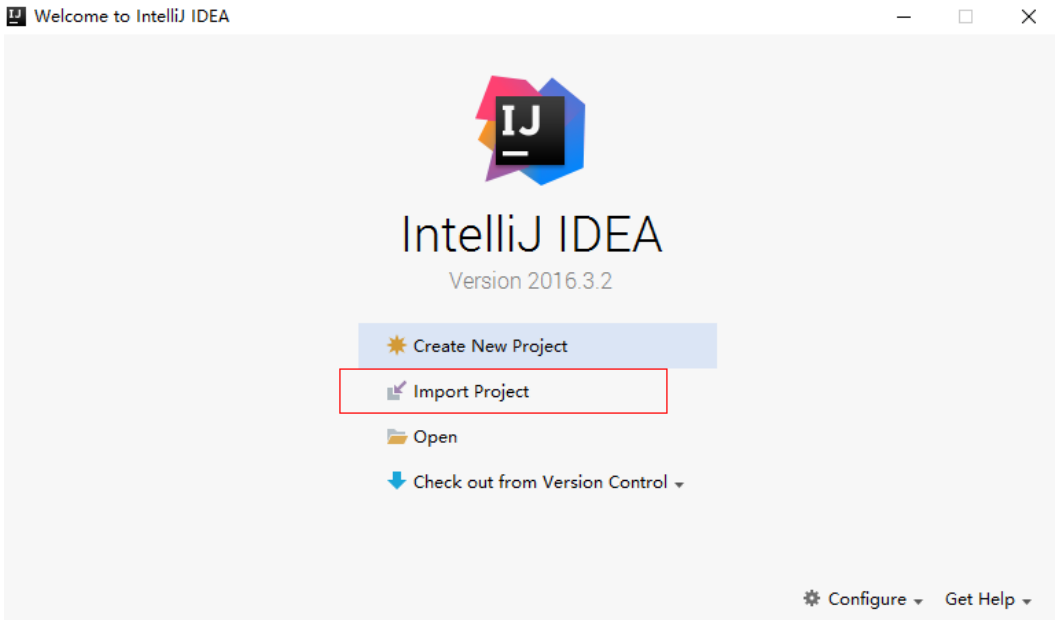


2. After the download is complete, run the installation file and install IntelliJ IDEA as prompted.

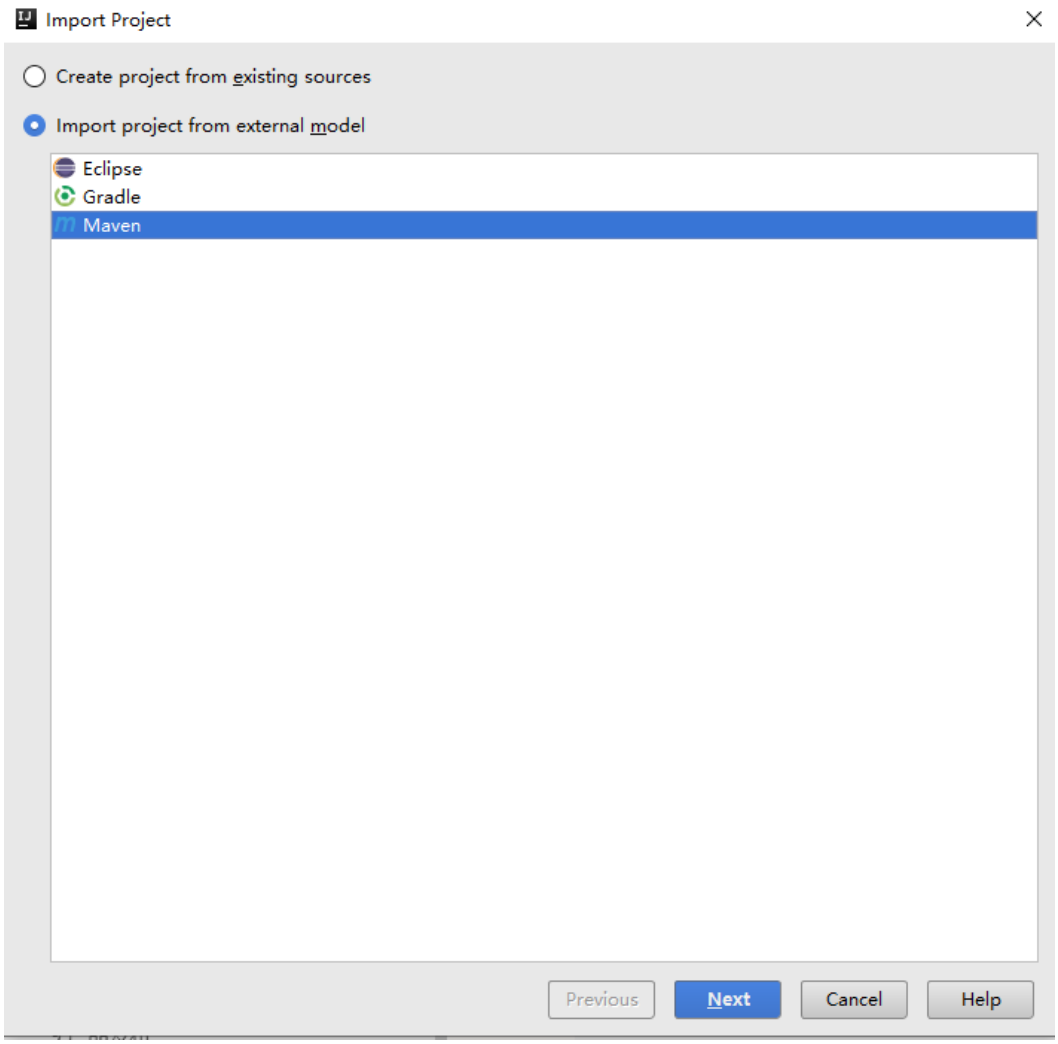
Importing Sample Code

Step 1 Download the **Java demo**.

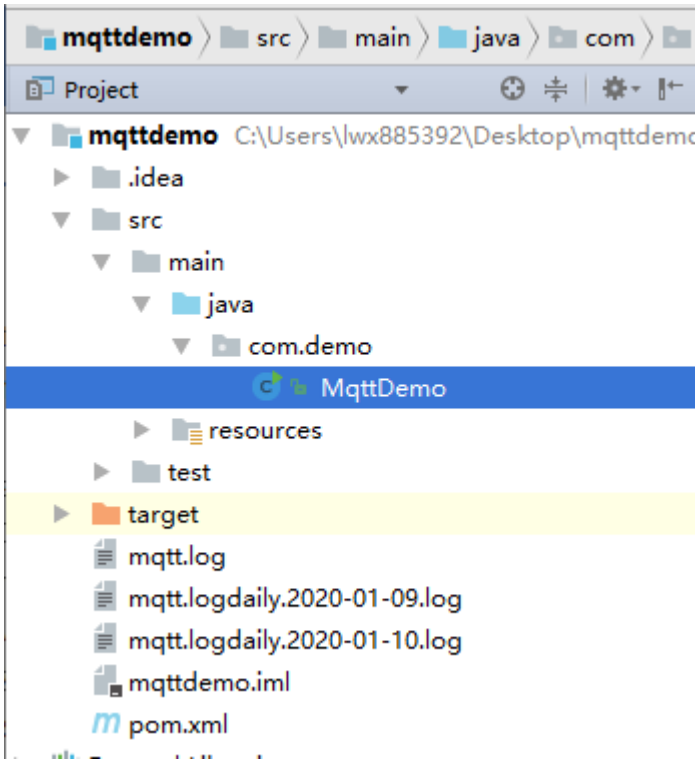
Step 2 Open the IDEA developer tool and click **Import Project**.



Step 3 Select the downloaded Java demo and click **Next**.



Step 4 Import the sample code.



----End

Establishing a Connection

Before you connect a device or gateway to the platform, establish a connection between the device or gateway and the platform by providing the device or gateway information.

1. Before establishing a connection, modify the following parameters:

```
// MQTT interconnection address of the platform
static String serverIp = "iot-mqtts.cn-north-4.myhuaweicloud.com";
// Device ID and secret obtained during device registration (Replace them with the actual values.)
static String deviceId = "722cb****************";
static String secret = "123456789";
```

   - **serverIp** indicates the device interconnection address of the platform. To obtain this address, see **Platform Interconnection 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());
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 stored in the **resources** directory of the demo is a certificate for verifying the platform identity. It is used for login authentication 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()));
```

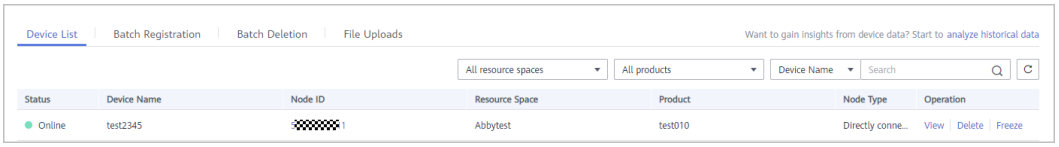
3. Call **client.connect(options, null, new IMqttActionListener())** to initiate a connection. The **MqttConnectOptions** object is passed.

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

4. The password passed by calling **options.setPassword()** is encrypted during creation of the **MqttConnectOptions** object. **getPassword()** is used to obtain the encrypted password.

```
public static String getPassword() {
    return sha256_mac(secret, getTimeStamp());
}
/* Call the SHA256 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) {
        LOGGER.info("Error HmacSHA256 ==========" + e.getMessage());
    }
    return passWord;
```

5. After the connection is established, the device becomes online.



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 released by the MQTT broker. Learn about preset topics of the platform in **Topic Definition**. For details about the API information, see **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**.

```
// ReportJSON 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 collects data. If this parameter is not specified, the system time is used by default.

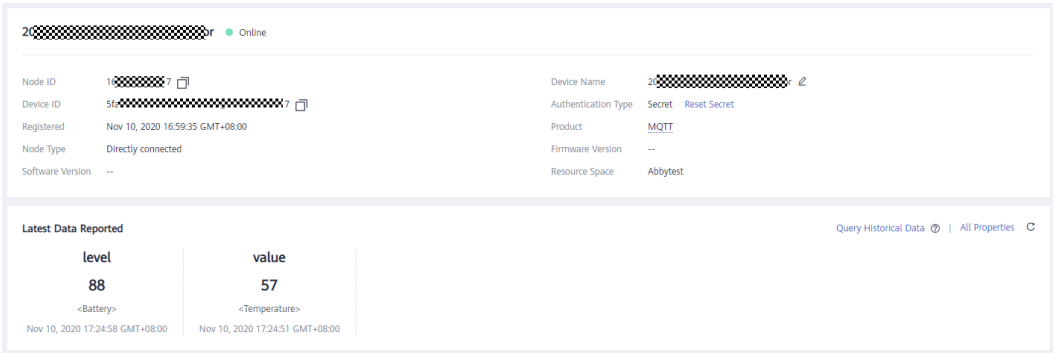
After a device or gateway is connected to the platform, you can call **MqttClient.publish(String topic,MqttMessage message)** to report device properties to the platform.

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 information, see **Reporting Device 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 API Reference** to connect MQTT devices to the platform. You can also use **MQTT.fx to experience device access** and verify whether devices can interact with the platform and publish or subscribe to messages.

4.3.4 Python Demo

Introduction

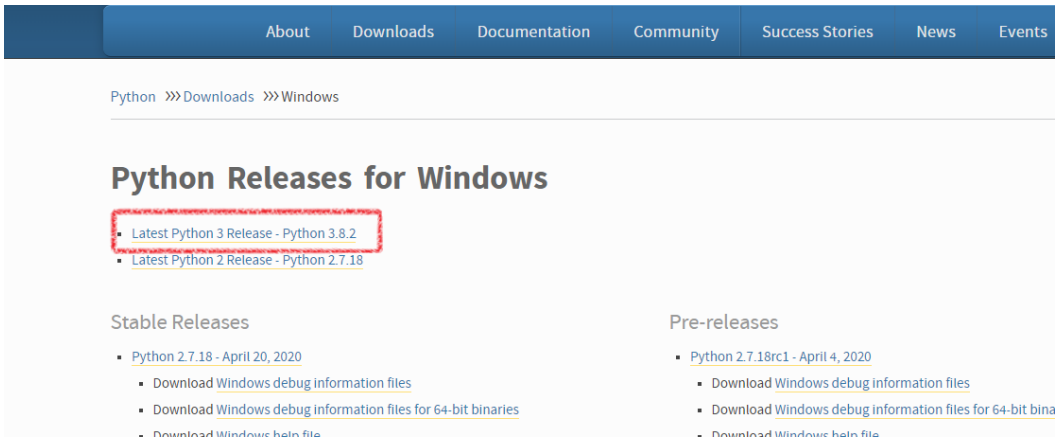
This section uses Python as an example to describe how to connect devices to the platform over MQTTS/MQTT and how to **report data** and **deliver commands** using **platform APIs**. For details about device access in other languages, see **Obtaining Resources**.

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

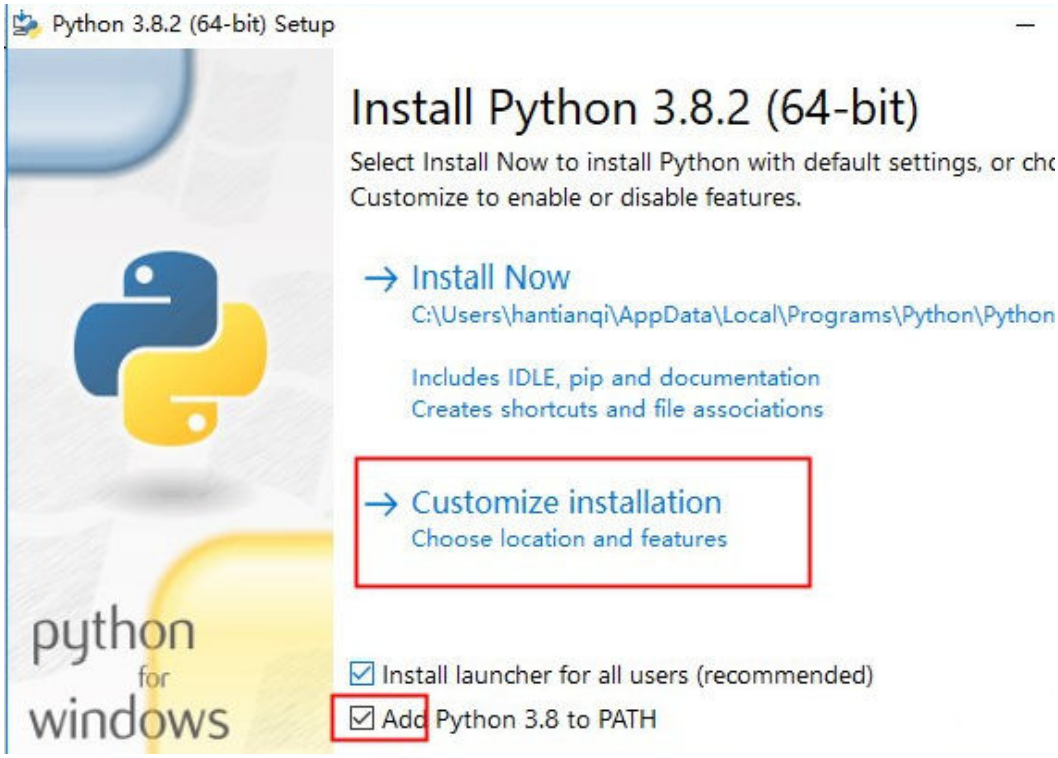
Preparations

- Installing Python
  a. Go to the **Python website** to download and install a desired version. (This procedure 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.

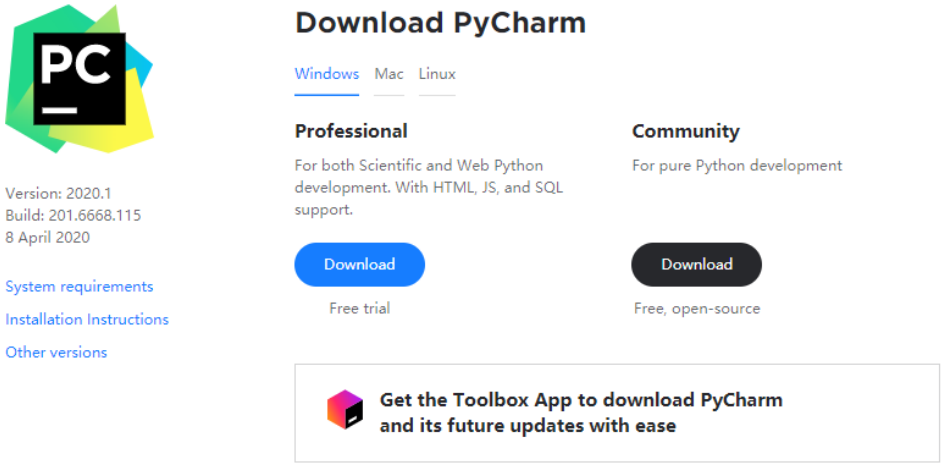
c. Select **Add python 3.8 to PATH** (if it is not selected, you need to manually configure environment variables), click **Customize installation**, and install Python as prompted.



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**.

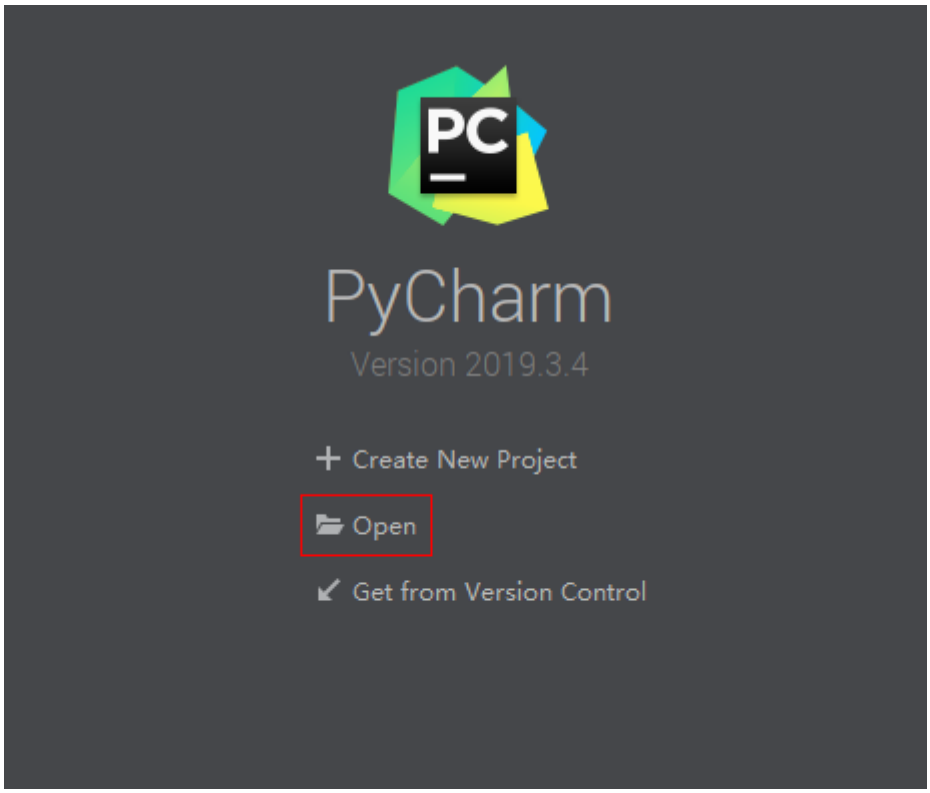


  The professional edition is recommended.
  b. Run the .exe file and install PyCharm as prompted.

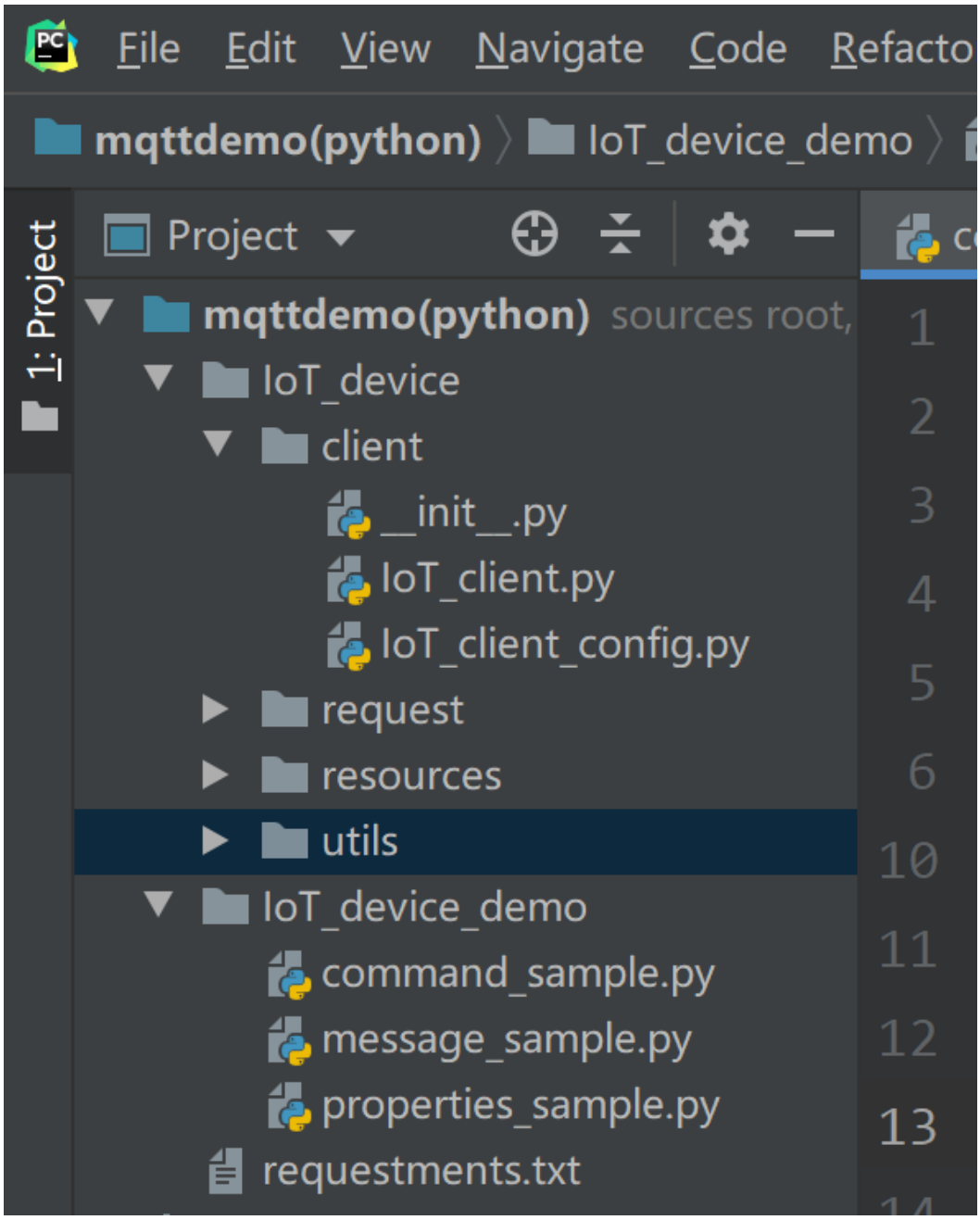
Importing Sample Code

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 reporting properties

- **IoT_device/client**: Used for paho-mqtt encapsulation

  IoT_client_config.py: Client configurations, such as the device ID and secret.

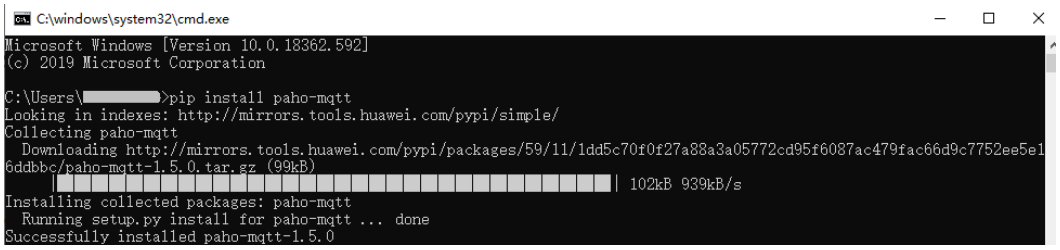
  IoT_client.py: MQTT-related function configurations, such as connection, subscription, release, and response.

- **IoT_device/Utils**: Tool methods, such as obtaining the timestamp and encrypting a secret
- **IoT_device/resources**: Stores 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:

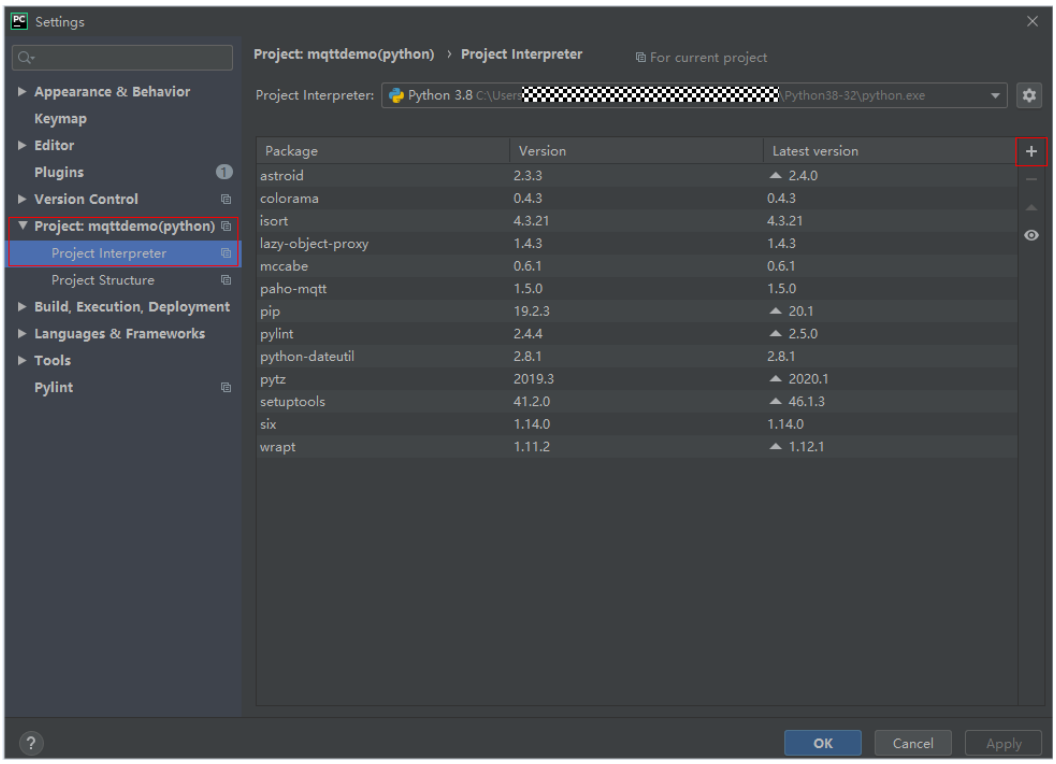
- Method 1: Use the pip tool to install paho-mqtt in the CLI. (The tool is already provided when installing Python.)

  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.

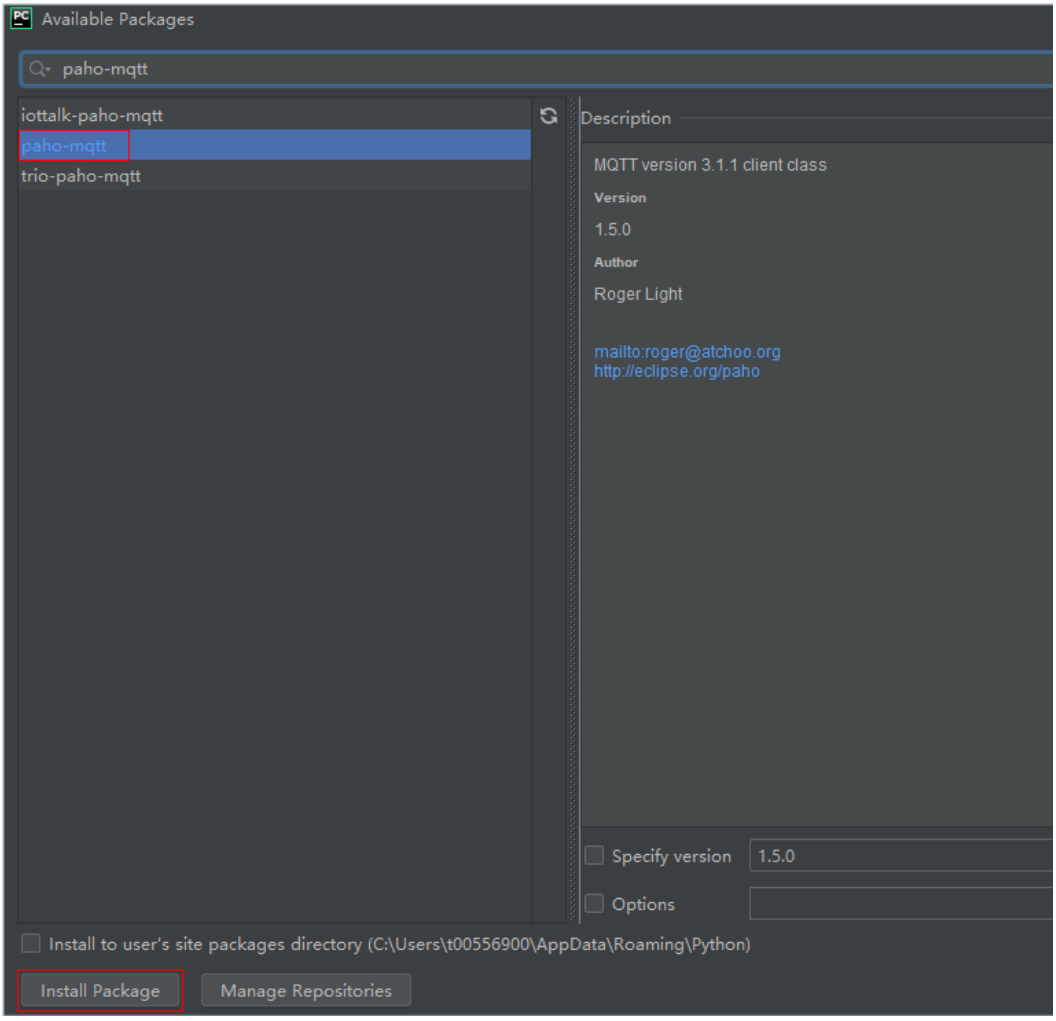


- Method 2: Install paho-mqtt using PyCharm.

  a. Open PyCharm, choose **File** > **Setting** > **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

Before you connect a device or gateway to the platform, establish a connection between the device or gateway and the platform by providing the device or gateway information.

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='123456789', is_ssl=True)
# Create a device.
iot_client = IotClient(client_cfg)
```

   - **server_ip**: Indicates the device interconnection address of the platform. To obtain this address, see **Platform Interconnection 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**: Obtain the values 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 released by the MQTT broker. Learn about preset topics of the platform in **Topic Definition**.

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 Command Delivery

The **command_sample.py** file provides the function of responding to commands delivered by the platform. For details about the API information, see **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

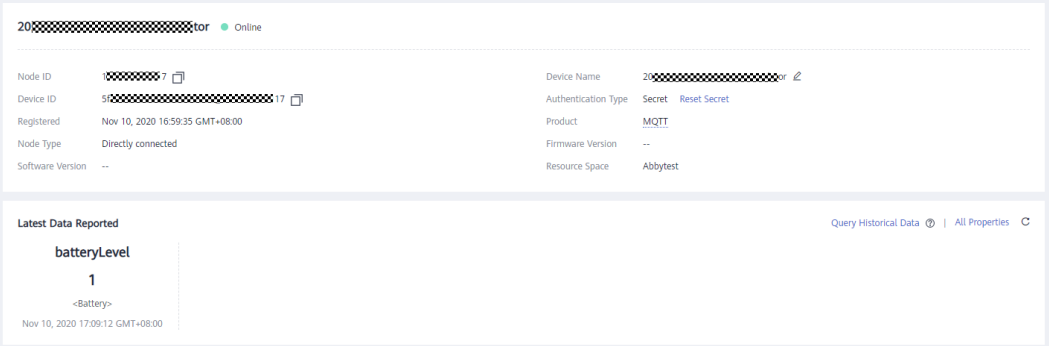
Devices can report their properties to the platform. For details, see **Reporting Device Properties**.

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_propertis.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 Messages

Message reporting refers to 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
```

4.3.5 Android Demo

Overview

This section uses Android as an example to describe how to connect a device to the IoT platform over MQTTS or MQTT and how to use **platform APIs** to **report data** and **deliver commands**. For details on other programming languages, see **Device Development Resources**.

Prerequisites

- You have installed Android Studio. If not, install Android Studio by following the instructions provided on the **Android website** and then install **the JDK**.

- You have obtained the device access addresses from the **IoTDA console**. For details, see **Platform Connection Information**.
- You have created a product and device on the **IoTDA console**. For details, see **Creating a Product**, **Registering an Individual Device**, or **Registering a Batch of Devices**.

Preparations

- Install Android Studio.

  Go to the **Android website** to download and install a desired version. The following uses Android Studio 3.5 running on 64-bit Windows 10 as an example.

  Android Studio downloads

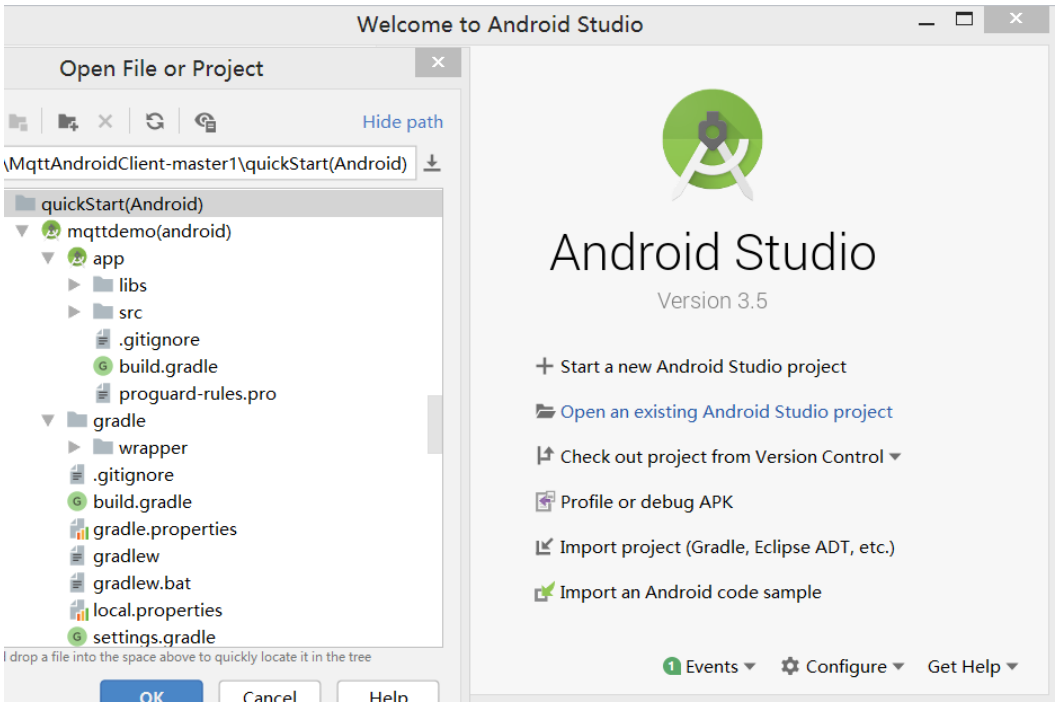
Platform	Android Studio package	Size	SHA-256 checksum
Windows (64-bit)	android-studio-ide-192.6392135-windows.exe Recommended	756 MB	07b6df807fda59e69f05b85ff6f6bd0c70d09e57fb151197155ef5f115f96e59
	android-studio-ide-192.6392135-windows.zip No .exe installer	770 MB	24f8f9ce467b935c25d89b90cad402d21dd45d4ba9af1ad35baeeb414609e483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

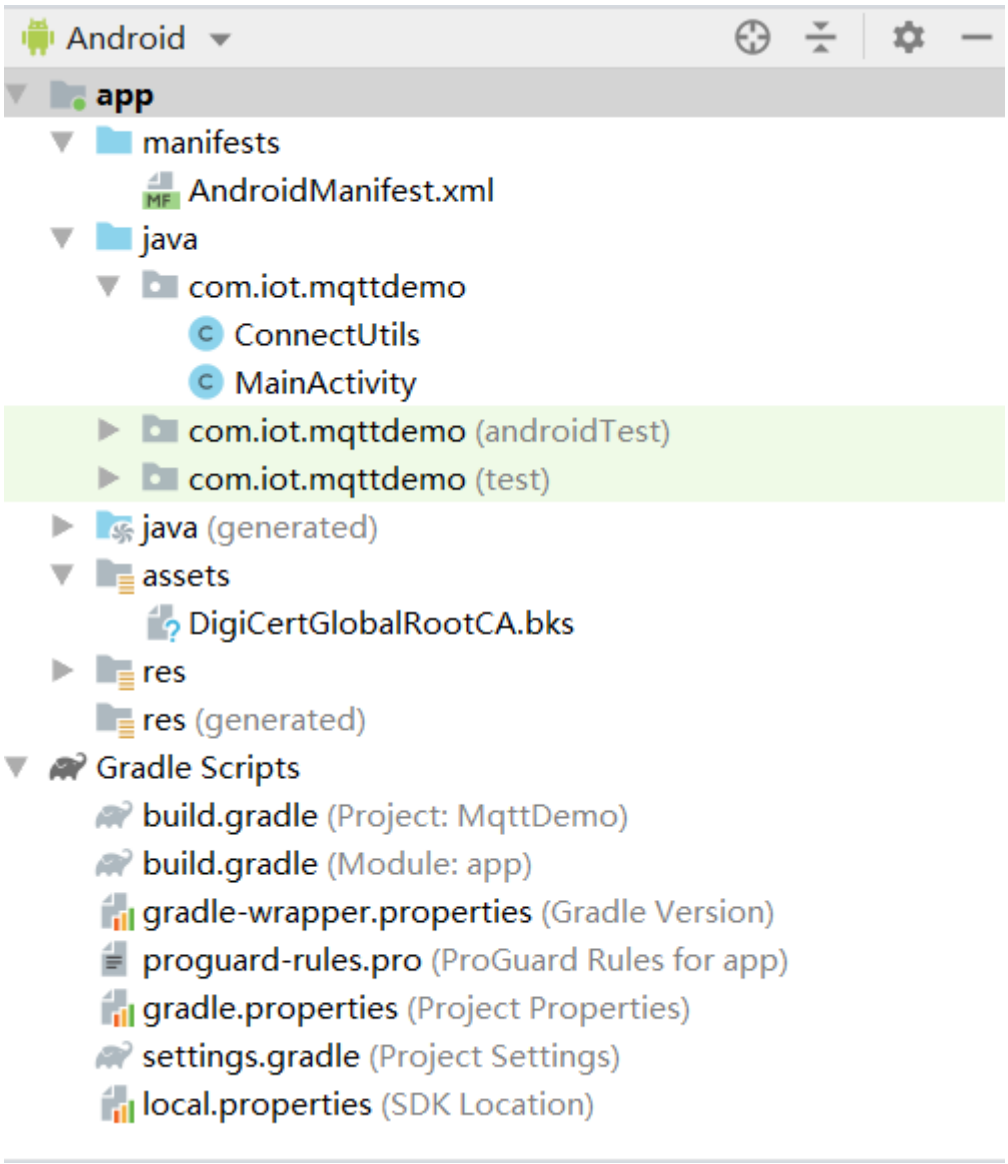
Importing Sample Code

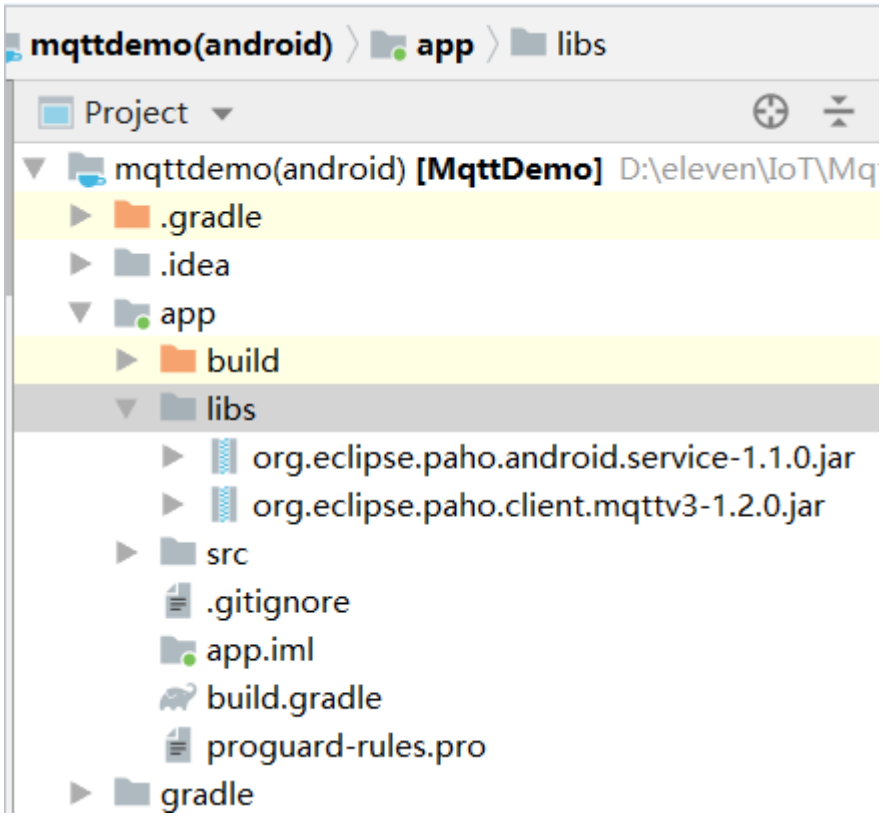
Step 1 Download the sample code **quickStart(Android)**.

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.

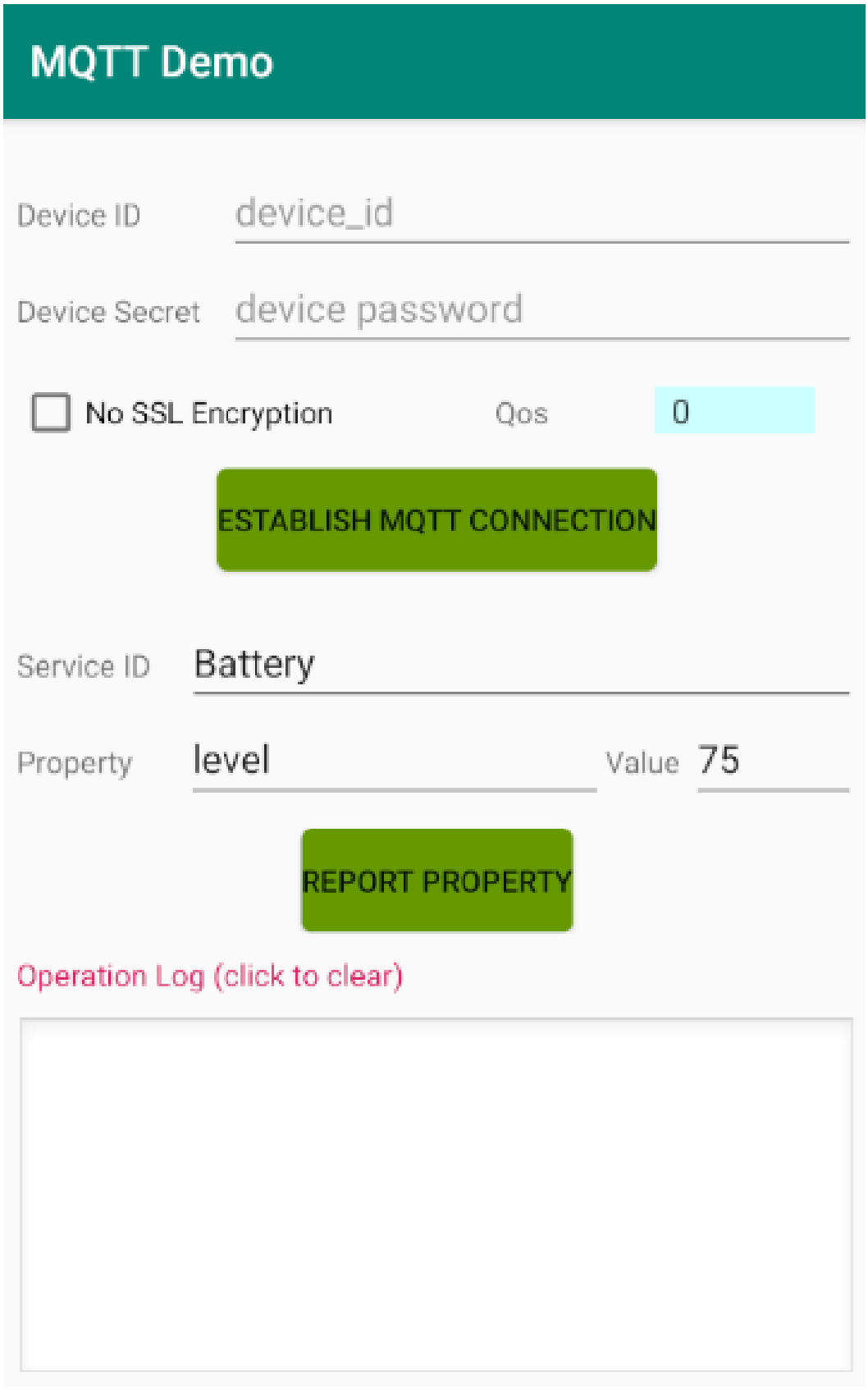
```
<service android:name="org.eclipse.paho.android.service.MqttService" />
```

- **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, enter the domain name for device access. (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_connet_ssl.setText ("SSL encryption");
        } else {
            isSSL = false;
            checkbox_mqtt_connet_ssl.setText ("no SSL encryption");
        }
    }
})
```

Establishing a Connection

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

1. Call the **MainActivity** class to establish an MQTT or MQTTS connection. By default, MQTT uses port 1883, and MQTTS uses port 8883 (a certificate must be loaded).
```
if (isSSL) {
    editText_mqtt_log.append("Starting to establish an MQTTS connection" + "\n");
    serverUrl = "ssl://" + IOT_PLATFORM_URL + ":8883";
} else {
    editText_mqtt_log.append("Starting to establish an MQTT connection" + "\n");
    serverUrl = "tcp://" + IOT_PLATFORM_URL + ":1883";
}
```
2. Call the **getMqttsCerificate** method in the **ConnectUtils** class to load an SSL certificate. This step is required only if an MQTTS connection is established.

   The **DigiCertGlobalRootCA.bks** file is used to verify the platform identity 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 released by the broker. **Topic Definition** describes preset topics of the platform.

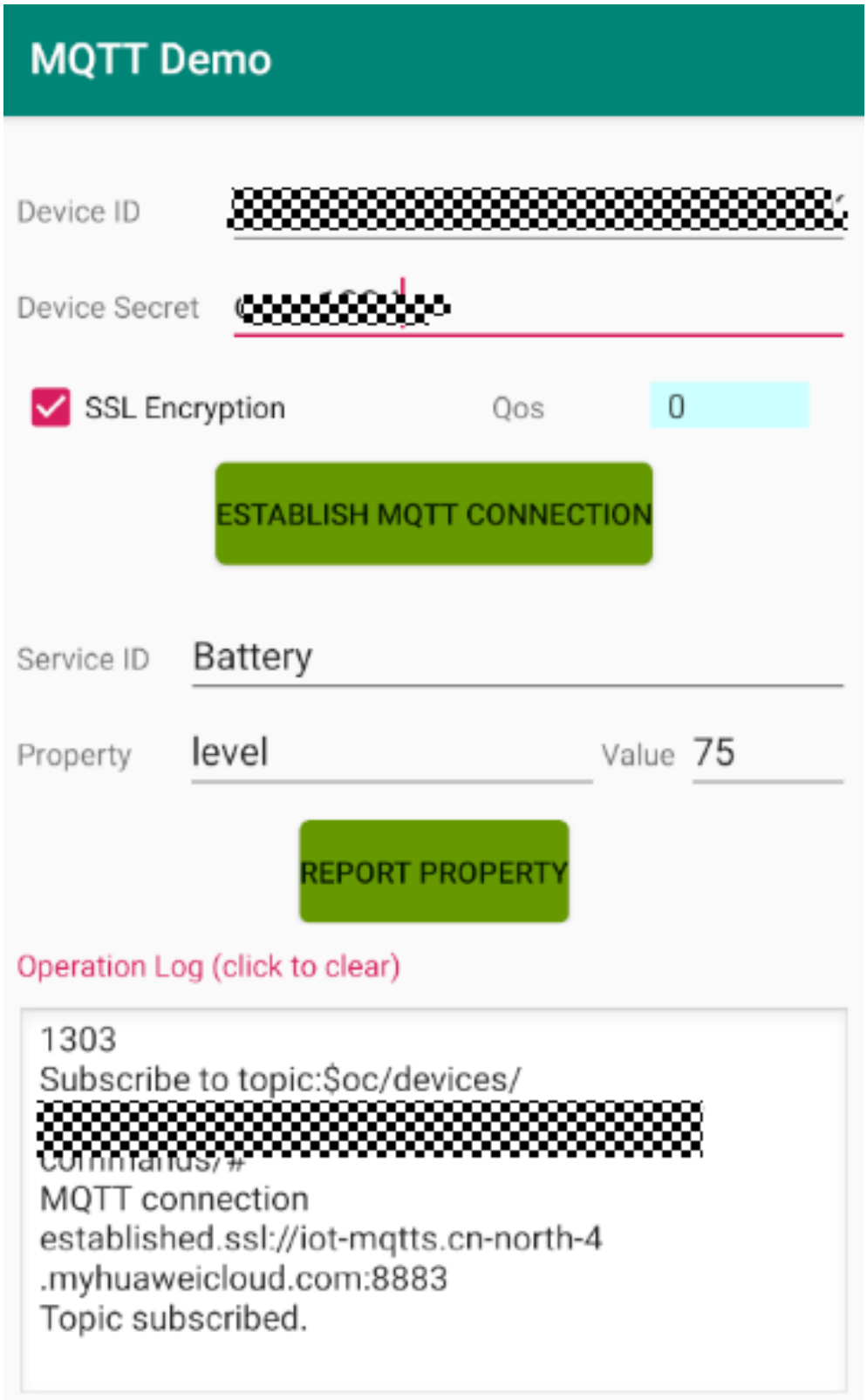
The **MainActivity** class provides the methods for delivering subscription commands to topics, subscribing to topics, and unsubscribing from topics.

```
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:



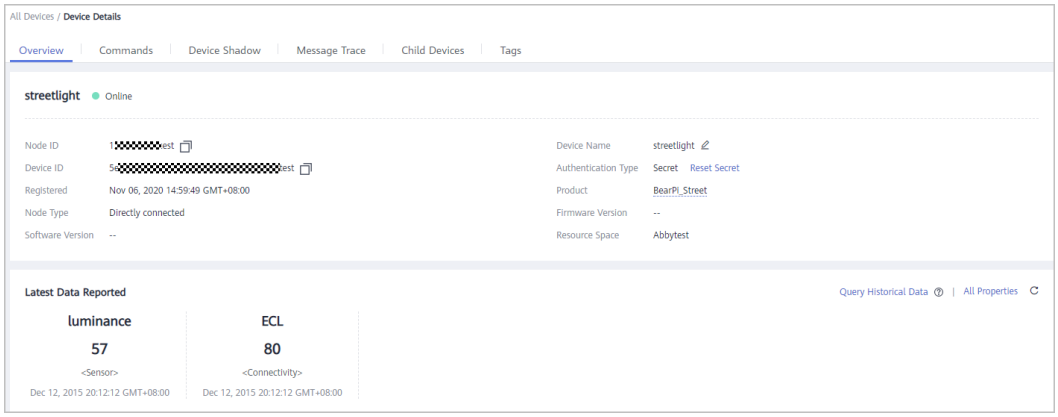
Reporting Properties

A device reports its properties to the platform. For details on the API, see **Reporting Device 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.

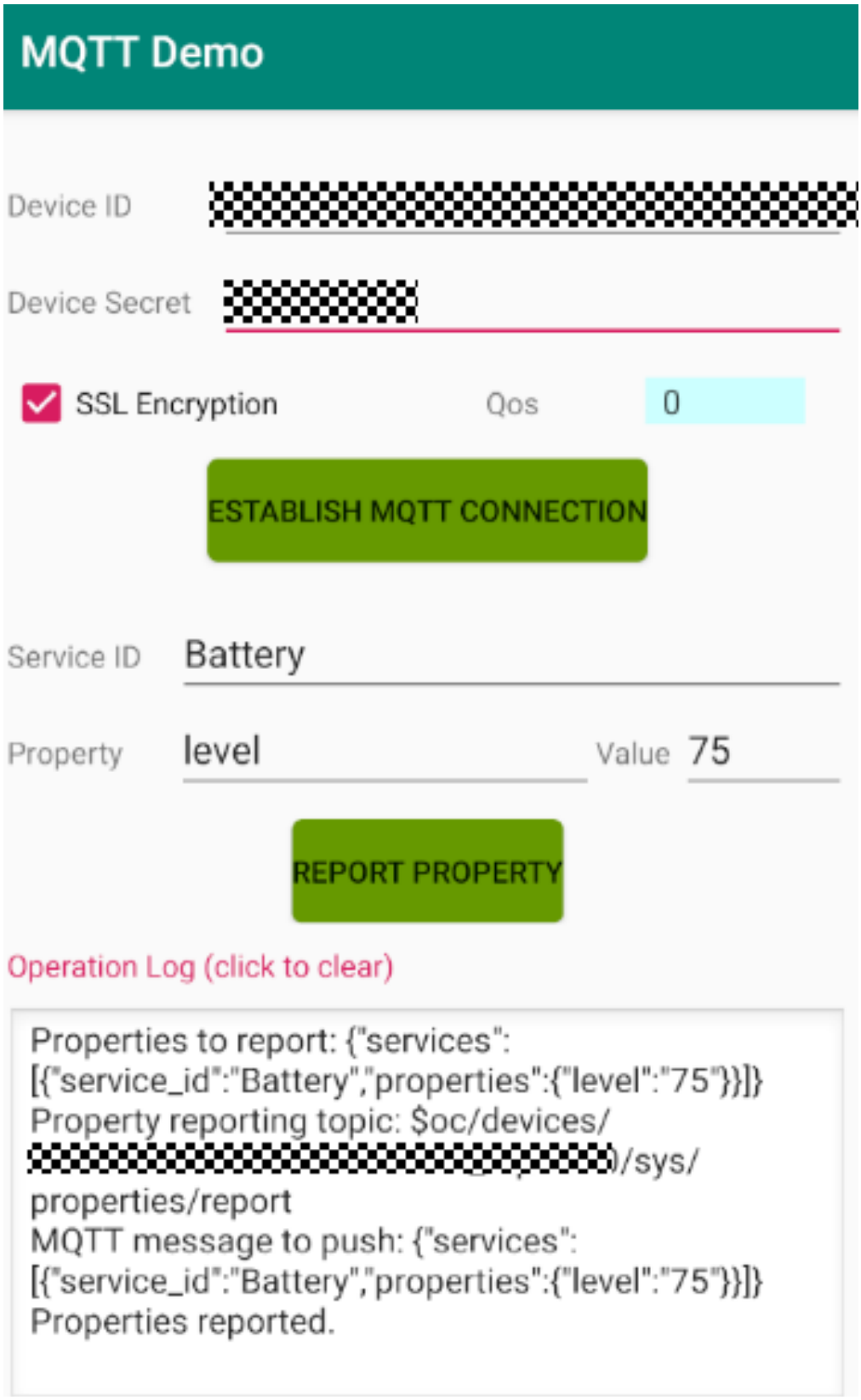
Receiving Commands

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 function.

```
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:



4.3.6 C Demo

Overview

This section uses C as an example to describe how to connect a device to the IoT platform over MQTTS or MQTT and how to use **platform APIs** to **report data** and **deliver commands**. For details on other programming languages, see **Device Development Resources**.

Prerequisites

- You have installed the Linux operating system (OS) is used and GCC (4.8 or later).
- You have obtained OpenSSL (required in MQTTS scenarios) and Paho library dependencies.
- You have obtained the device access addresses from the **IoTDA console**. For details, see **Platform Connection Information**.
- You have created a product and device on the **IoTDA console**. For details, see **Creating a Product**, **Registering an Individual Device**, or **Registering a Batch of Devices**.

Preparations

- Compile 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, in the directory **/home/test**), 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
  

  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.

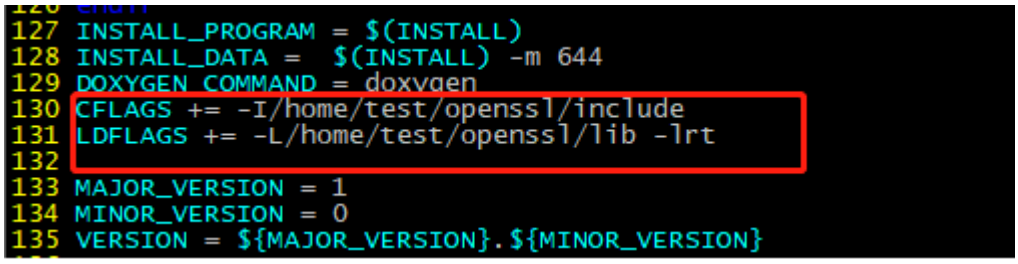

```
make install
```

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.

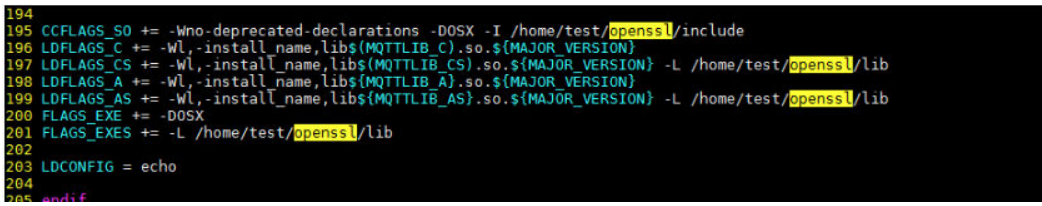
Copy these files to the **lib** folder of the demo and copy the content in **/home/test/openssl/include/openssl** to **include/openssl** of the demo.

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.

- Compile 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. Display the number of rows.
     ```
     :set nu
     ```
     iii. Add the following two lines (customized OpenSSL header files and library files) after line 129:
     ```
     CFLAGS += -I/home/test/openssl/include
     LDFLAGS += -L/home/test/openssl/lib -lrt
     ```



     iv. Change the addresses in lines 195, 197, 199, and 201 to the corresponding addresses.

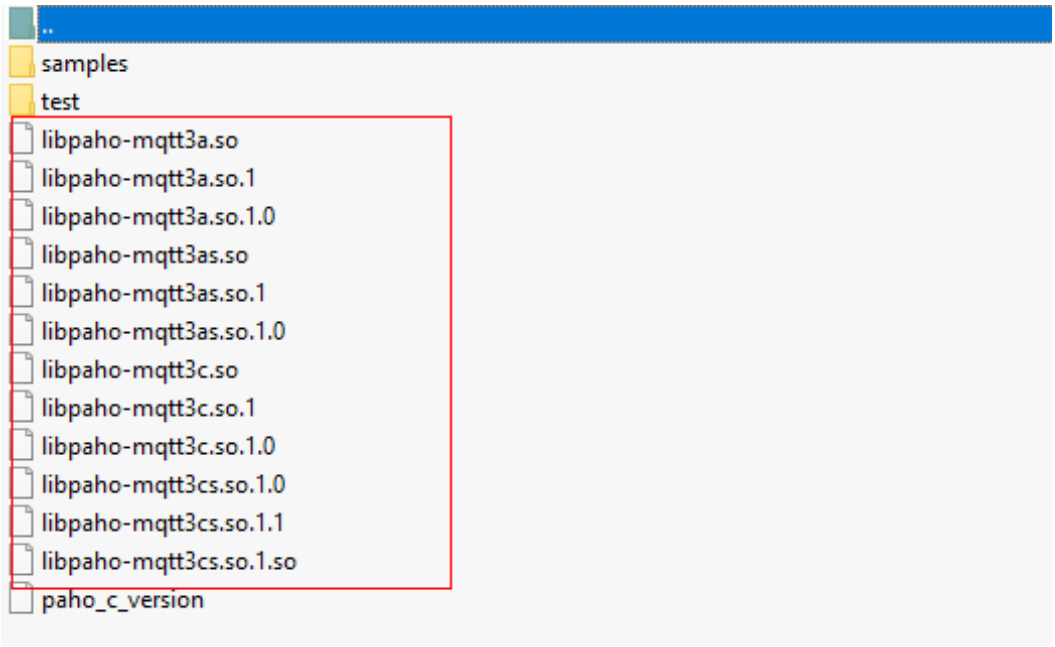


  d. Start the compilation.

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.



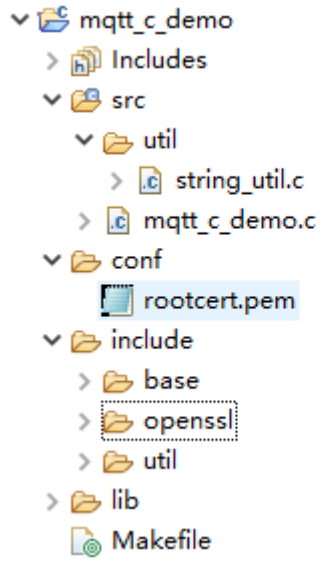
f. Copy the Paho library file.

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.

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: tool resource file
- **conf**: certificate directory

  rootcert.pem: certificate used by the device to verify the platform identity. It is used for login authentication when the device connects to the platform.
- **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.

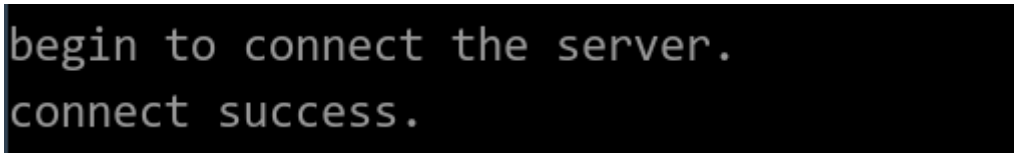
```
char *uri = "ssl://iot-mqtts.cn-north-4.myhuaweicloud.com:8883";
int port = 8883;
char *username = "5ebac693352cfb02c567ec88_test2345"; //deviceId
//char *username = "test6789";
char *password = "602d6cc77d87271be8f462f52d27d818";
```

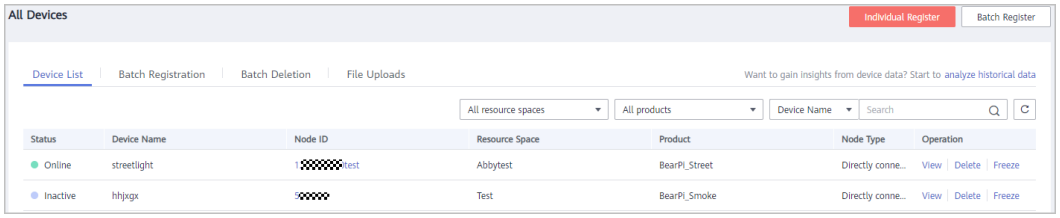
   Note: MQTTS uses port 8883 for access. If MQTT is used for access, the URL is **tcp://iot-mqtts.cn-north-4.myhuaweicloud.com:1883** and the port is **1883**.

2. Start the connection.
   - Run the **make** command to perform 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.





If the connection fails, the mqtt_connect_failure function executes backoff reconnection. Sample code:

```
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 released by the broker. **Topic Definition** describes preset topics of the platform.

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.

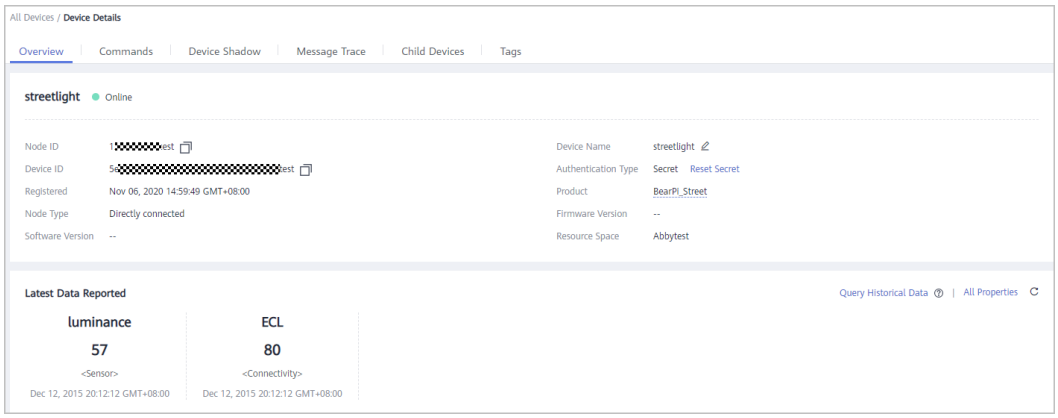
Reporting Properties

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.



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 Commands

After subscribing to a command topic, you can deliver a synchronous command on the console. For details, see **Synchronous Command Delivery to MQTT Devices**.

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=b5fb4352-43
cb-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
}
```

4.3.7 C# Demo

Overview

This section uses C# as an example to describe how to connect a device to the IoT platform over MQTTS or MQTT and how to use **platform APIs** to **report data** and **deliver commands**. For details on other programming languages, see **Device Development 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 addresses from the **IoTDA console**. For details, see **Platform Connection Information**.
- You have created a product and device on the **IoTDA console**. For details, see **Creating a Product**, **Registering an Individual Device**, or **Registering a Batch of Devices**.

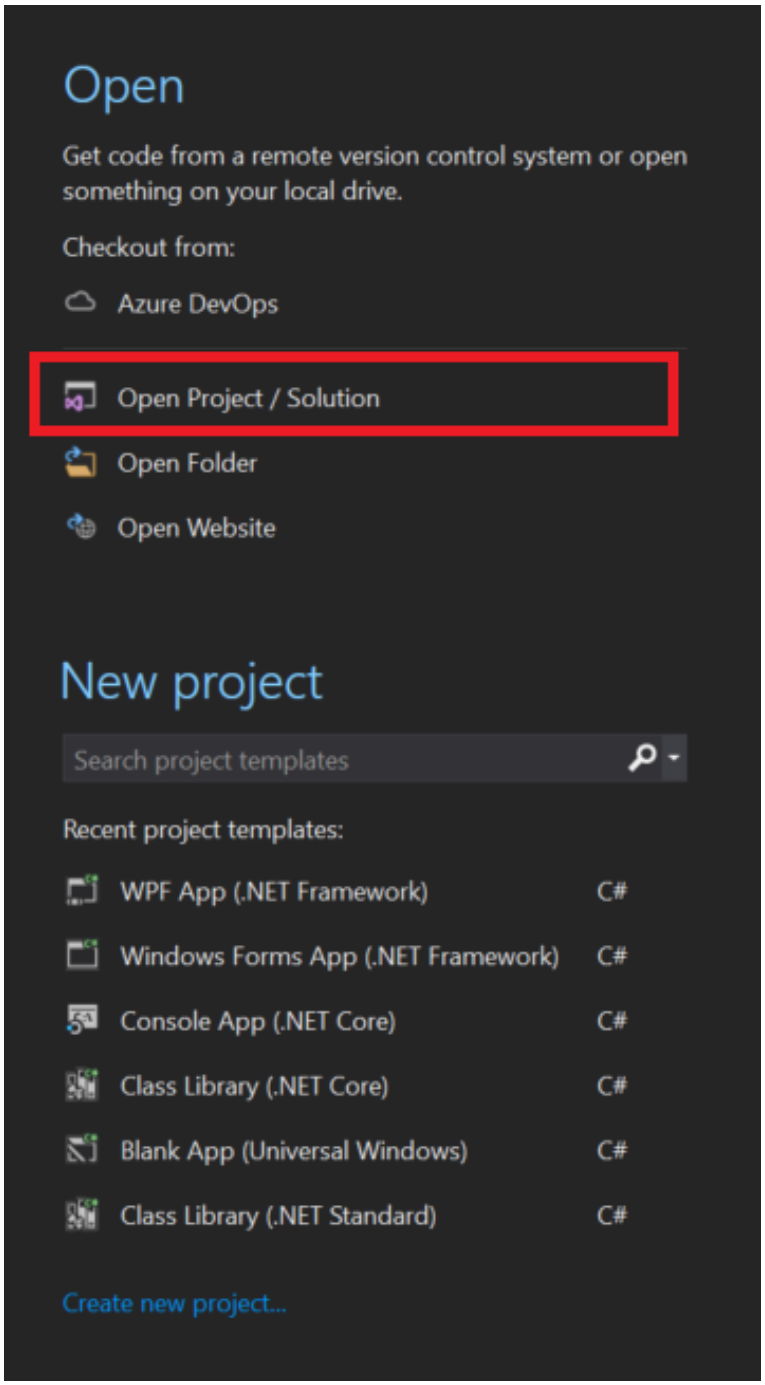
Preparations

- Go to the **Microsoft website** to download and install Microsoft Visual Studio of a desired version. (This document 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.

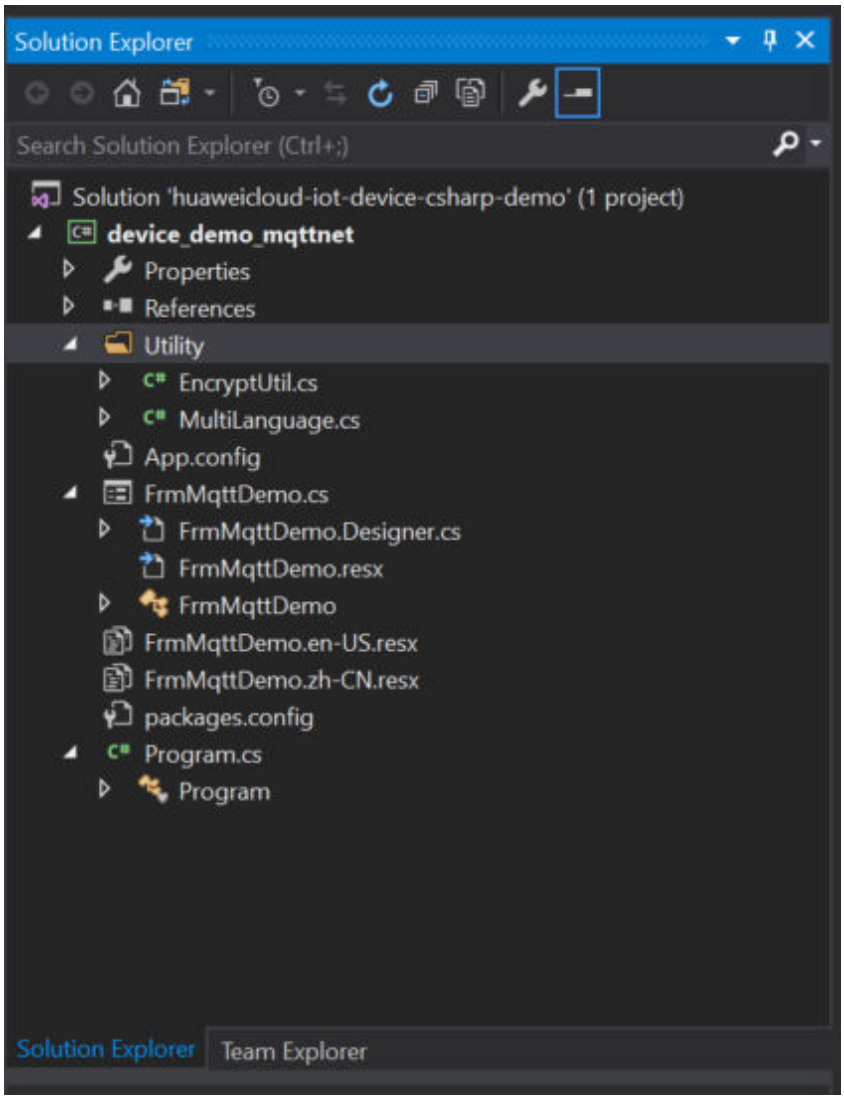
Importing Sample Code

Step 1 Download the sample code **quickStart(C#)**.

Step 2 Run Microsoft Visual Studio 2017, click **Open Project/Solution**, and select the sample downloaded.



Step 3 Import the sample code.



Description of the directories:

- **App.config**: server address and device information configuration file
- **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 .NET open-source library based on MQTT communications. It supports both MQTT servers and clients. The reference library file contains **MQTTnet.dll.**
  MQTTnet.Extensions.ManagedClient: v3.0.11 is an extended library that uses MQTTnet to provide additional functions for the managed MQTT client.

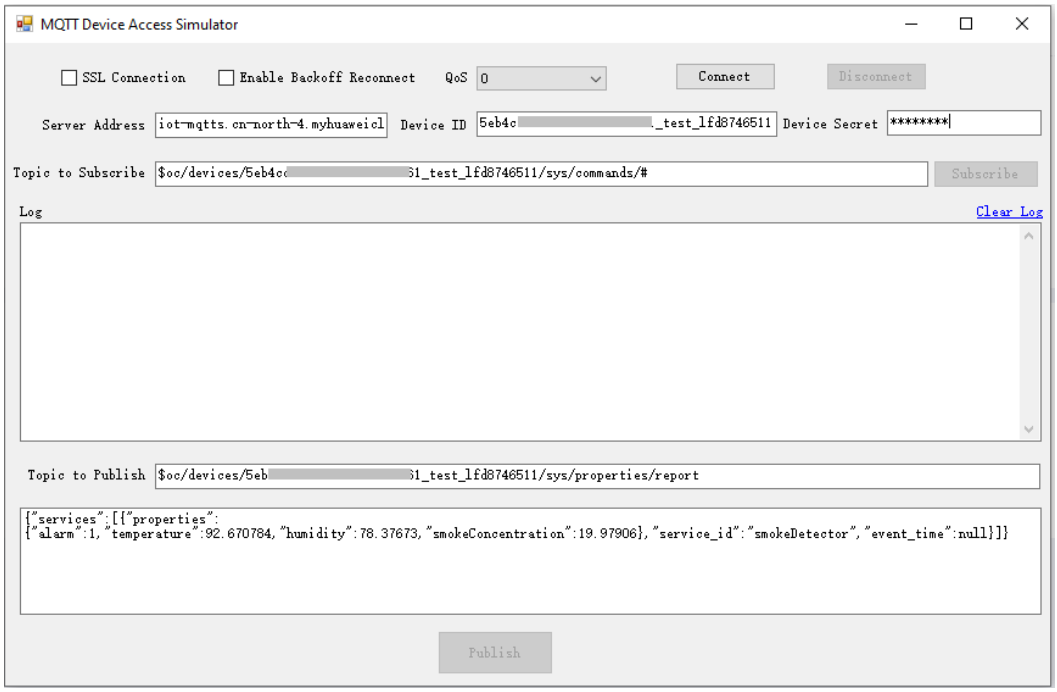
Step 4 Set the project parameters in the demo.

- **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: indicates the 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 this 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, QoS2 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 establish MQTT or MQTTS connections. By default, MQTT uses port 1883, and MQTTS uses port 8883. (In the case of MQTTS scenarios, you must load the **DigiCertGlobalRootCA.crt.pem** certificate for verifying the platform identity. The 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);
 })));
}
}
```

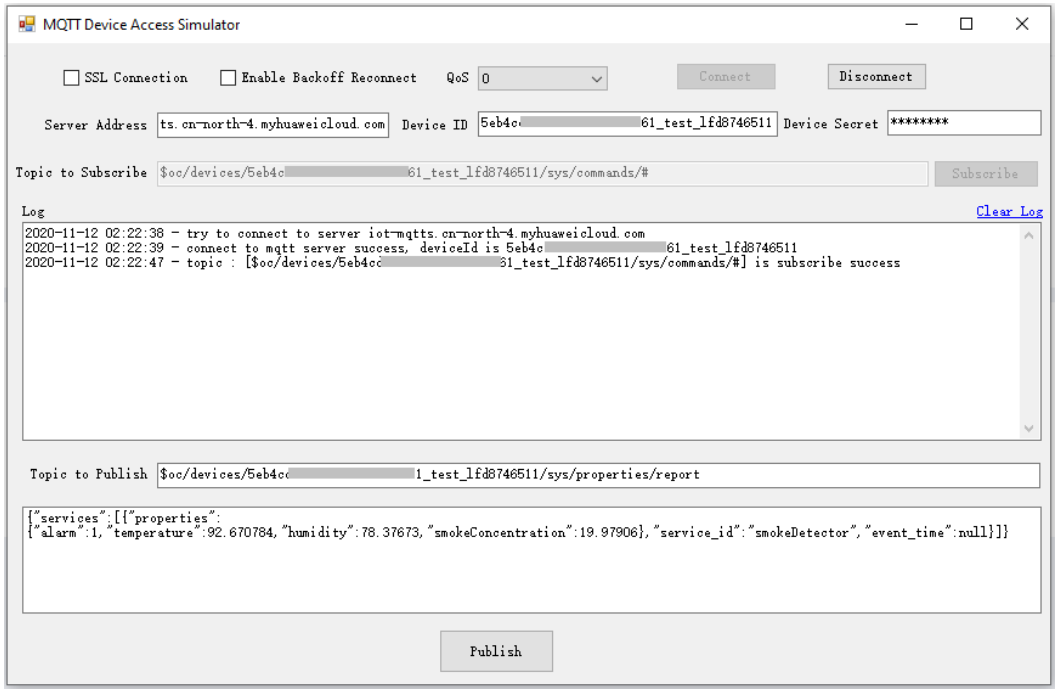
Subscribing to a Topic

Only devices that subscribe to a specific topic can receive messages about the topic released by the broker. **Topic Definition** describes preset topics of the platform.

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:

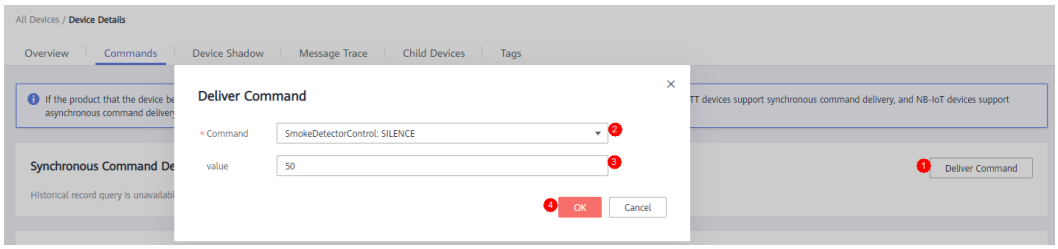


Receiving Commands

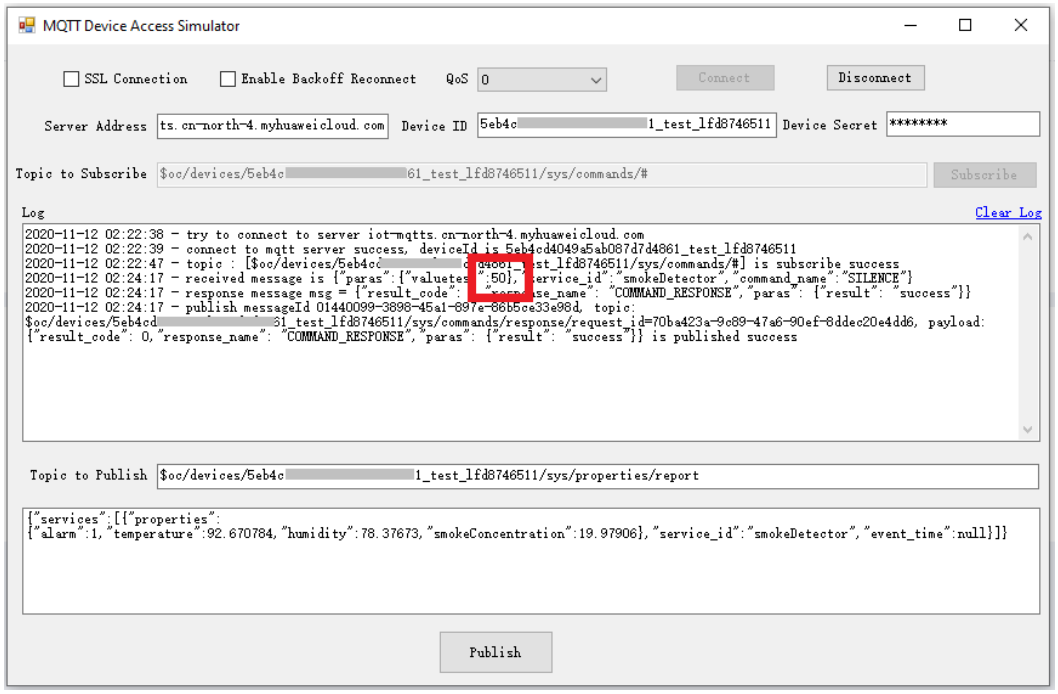
The **FrmMqttDemo** class provides the method for receiving commands delivered by the platform. After an MQTT connection is established and a topic is subscribed, you can deliver a command on the device details page of the **IoTDA console** or by using the **demo on the application side**. After the command is delivered, the MQTT callback function 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 **smokeDetector: SILENCE** and parameter value **50**.



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



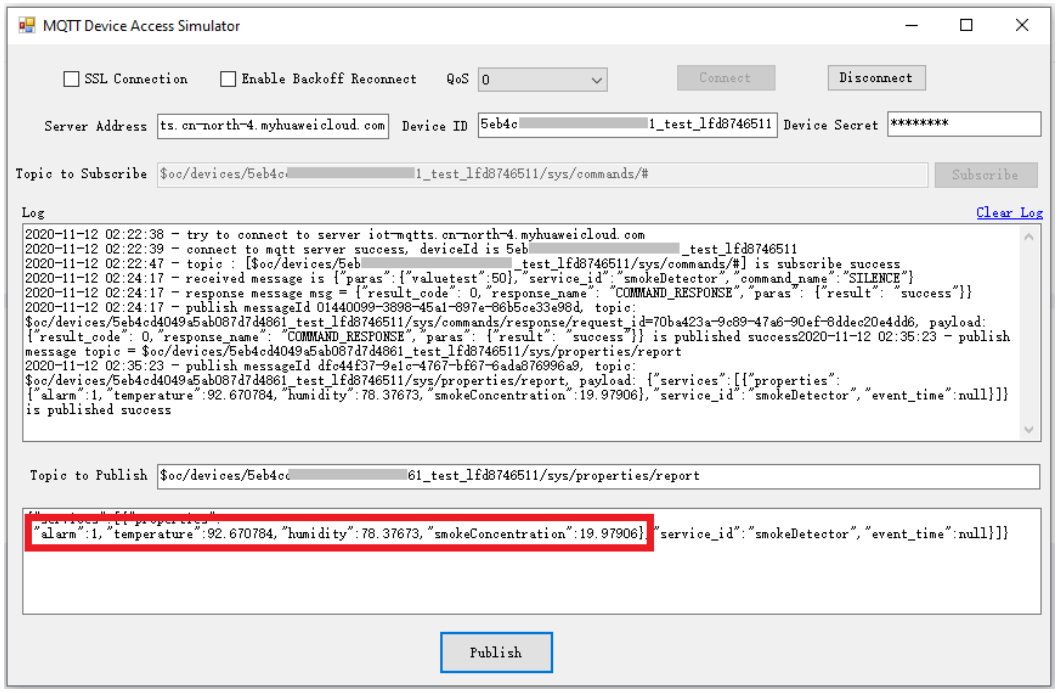
Publishing a Topic

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

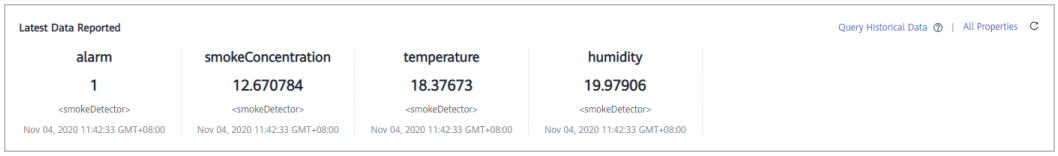
The **FrmMqttDemo** class implements the property reporting topic and property reporting.

```
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.

4.3.8 Node.js Demo

Overview

This section uses Node.js as an example to describe how to connect a device to the IoT platform over MQTTS or MQTT and how to use **platform APIs** to **report data** and **deliver commands**. For details on other programming languages, see **Device Development Resources**.

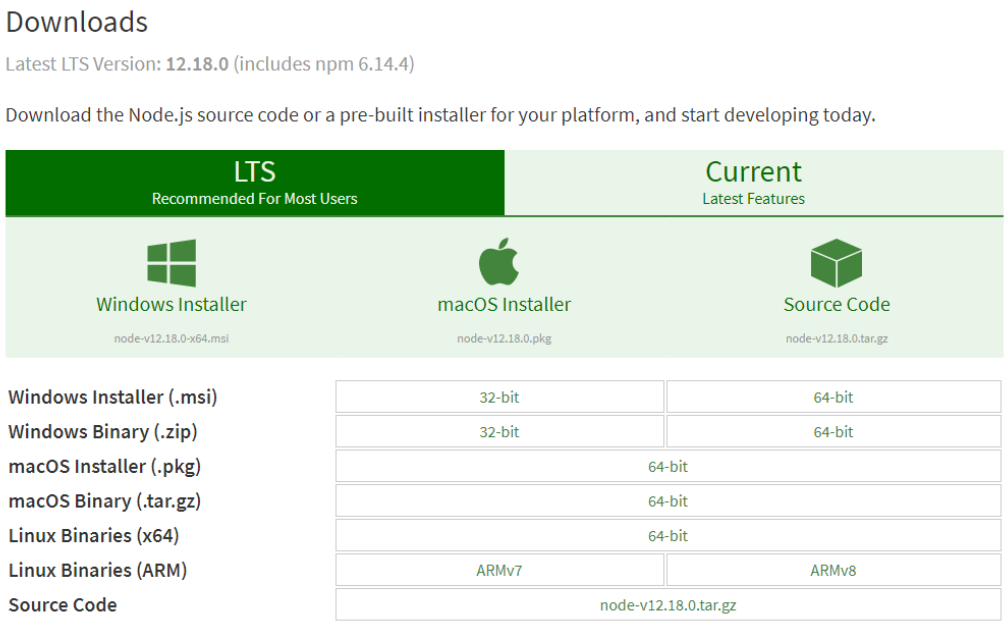
Prerequisites

- You have installed Node.js by following the instructions provided in **Install Node.js**.
- You have obtained the device access addresses from the **IoTDA console**. For details, see **Platform Connection Information**.

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

Preparations

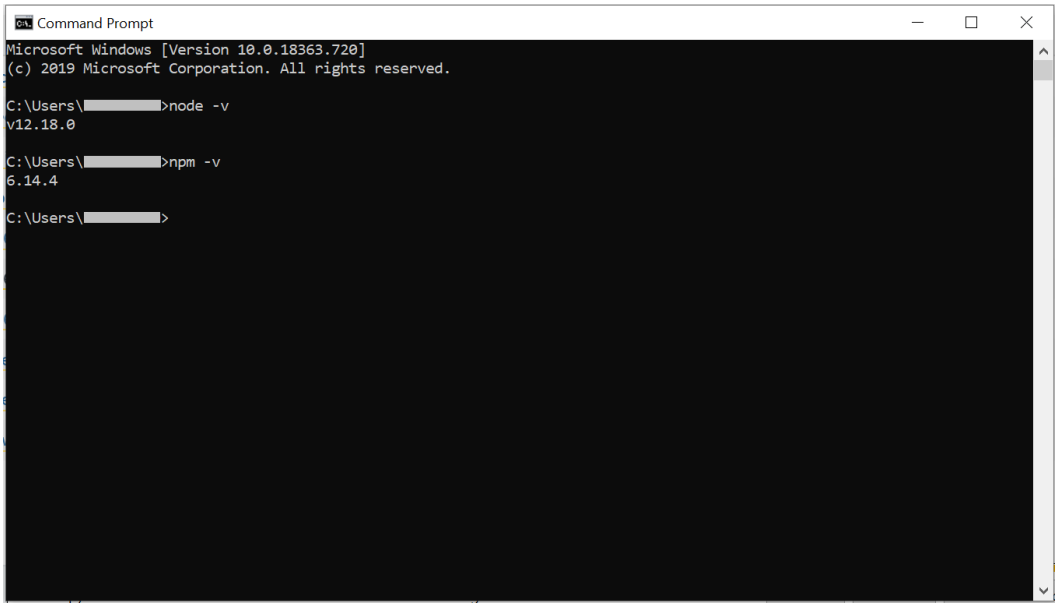
1. Go to the **Node.js website** to download and install a desired version. This document uses Windows 64-bit and Node.js v12.18.0 (npm 6.14.4) as an example.



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.

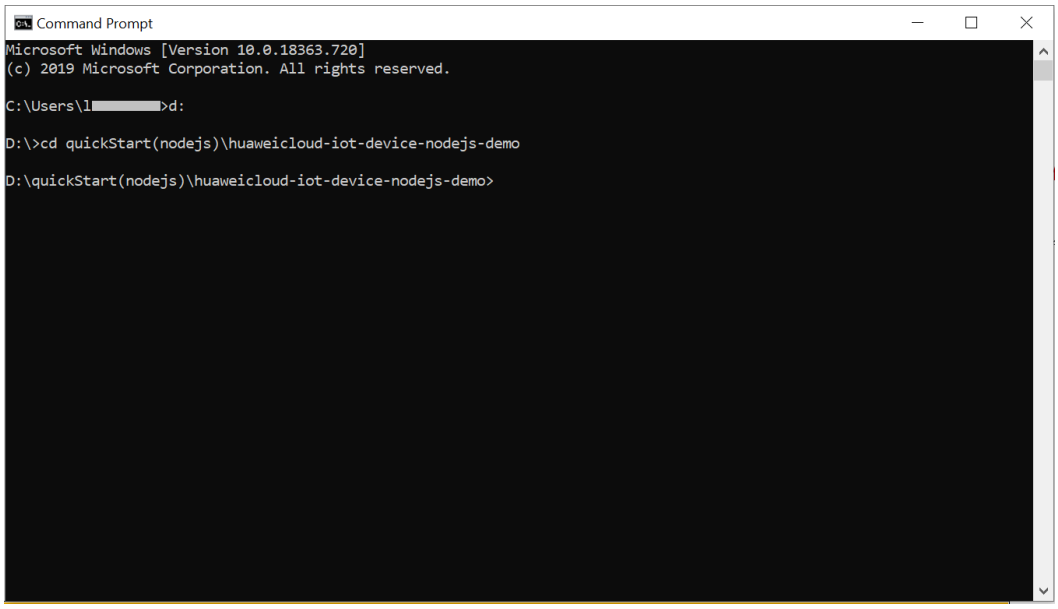
Step 2 Press **Win+r**, enter **cmd**, and press **Enter** to access the CLI. Run the following commands to install the global module:

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 encryption algorithm module.

npm install fs -g: This command is used to load the platform certificate.

Step 3 Find the directory where the file is decompressed.



Code directory:

- **DigiCertGlobalRootCA.crt.pem**: platform certificate file
- **MqttDemo.js**: Node.js source code for MQTT/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: indicates the 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 interconnection address of the platform
var serverUrl = "iot-mqtts.cn-north-4.myhuaweicloud.com";
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':'smokeDetector','event_time':null}]};
var responseReqJson = {'result_code': 0,'response_name': 'COMMAND_RESPONSE','paras': {'result': 'success'}};
```

Step 5 Select different options from mqtt.connect(options) to determine whether to perform SSL encryption during connection establishment on the device. You are advised to use the default MQTTS secure connection.

```
// Secure 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

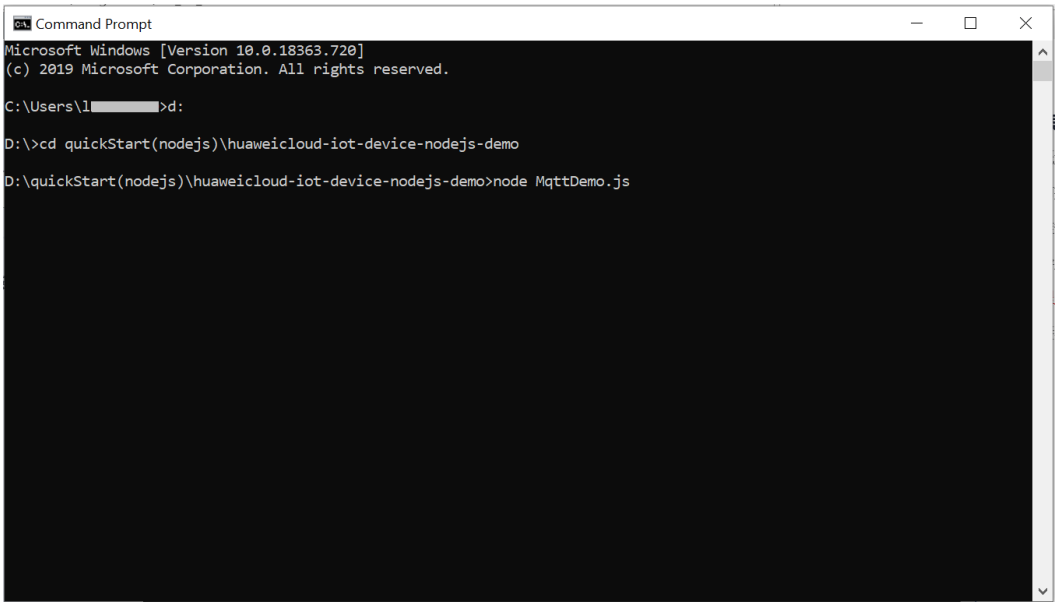
Starting the Demo

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

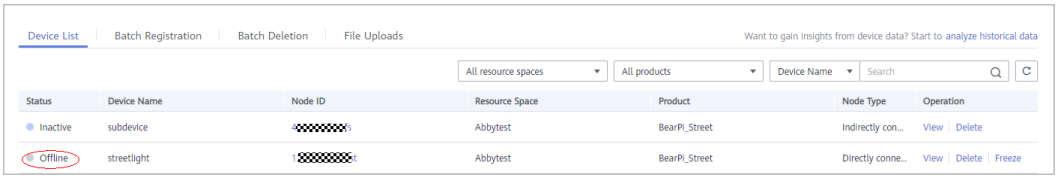
1. This demo provides methods such as establishing an MQTT or MQTTS connection. By default, MQTT uses port 1883, and MQTTS uses port 8883. (In the case of MQTTS connections, you must load the certificate for verifying the platform identity. The certificate is used for login authentication when the device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

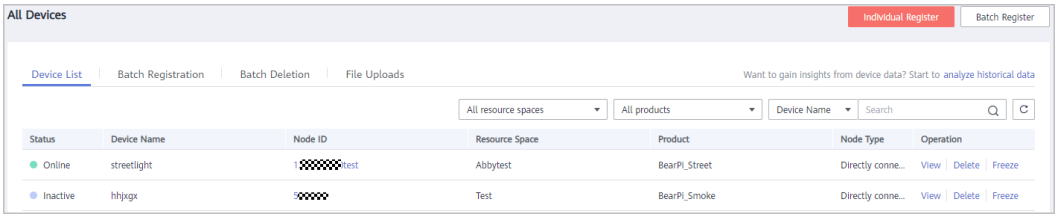
Find the Node.js demo source code directory, modify **key project parameters**, and start the demo.



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. Sample code:

```
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 released by the broker. **Topic Definition** describes preset topics of the platform. 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");
        }
    })
}
```

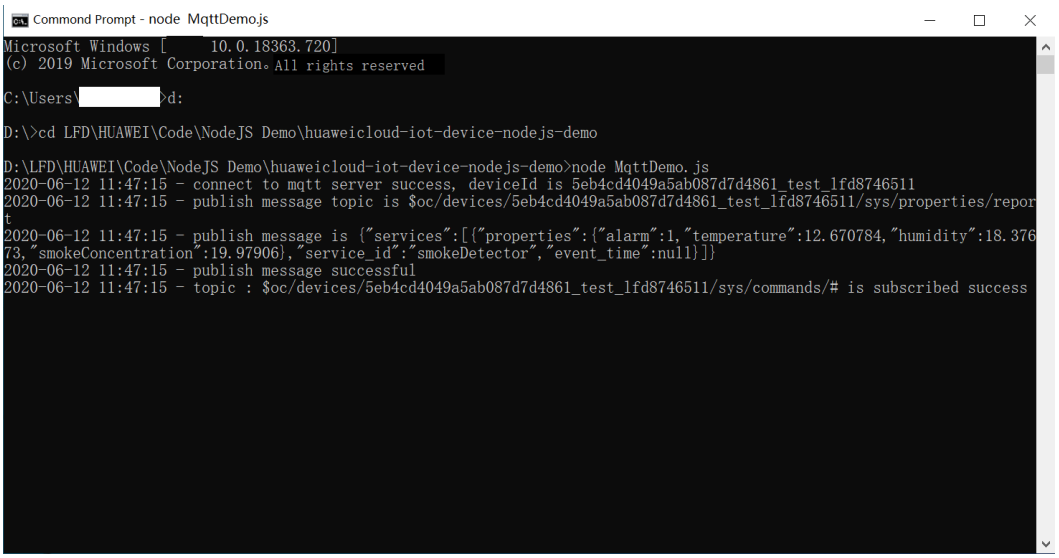
3. Publishing a topic means that a device proactively reports its properties or messages to the platform. For details, see the API **Reporting Device Properties**. After the connection is successful, call the **publishMessage** method to report properties.

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

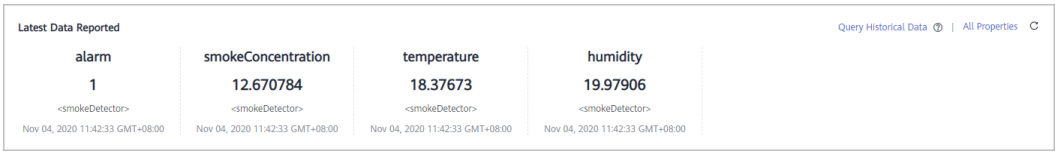
Reported properties in the JSON format:

```
var propertiesReportJson = {'services':[{'properties':{'alarm':1,'temperature':12.670784,'humidity':
18.37673,'smokeConcentration':19.97906},'service_id':'smokeDetector','event_time':null}]};
```

The following figure shows the CLI.



If the properties are reported, the following information is displayed on the IoTDA console:



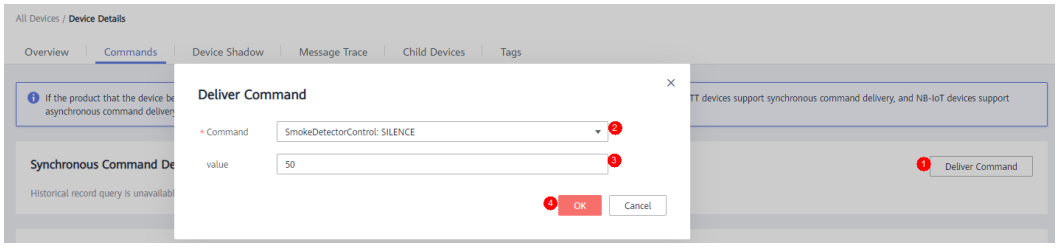
NOTE

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.

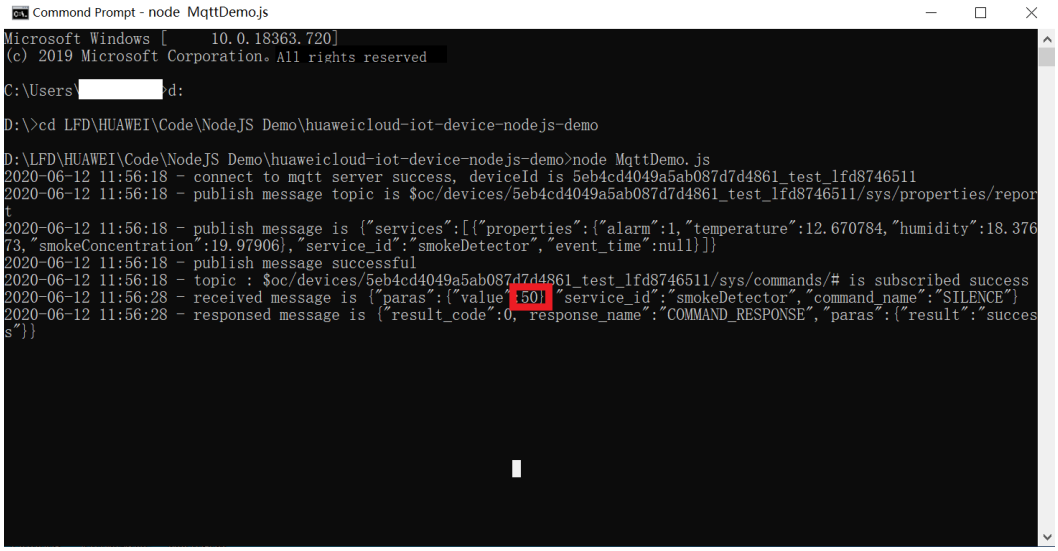
Receiving Commands

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.



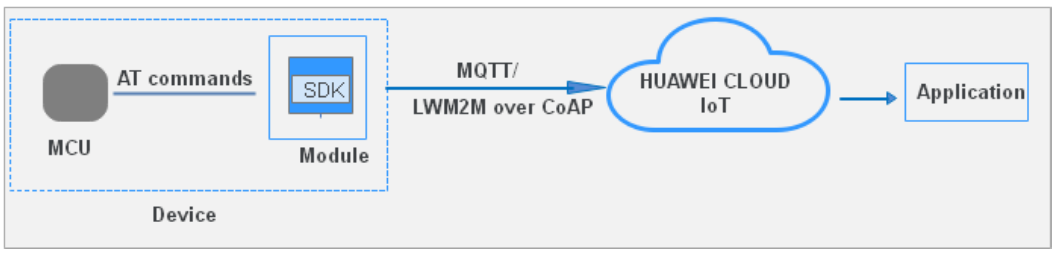
4.4 Using Huawei-Certified Modules for Access

Overview

Certified modules are pre-integrated with the **IoT Device SDK Tiny**. They have passed Huawei test, and comply with **Huawei's AT command** specifications. The following benefits are available for using Huawei-certified modules:

- Device manufacturers do not need to concern about how to connect to the HUAWEI CLOUD IoT platform on the MCU (for example, how to set the secret encryption algorithm and clientID composition mode during MQTT connection setup). To connect their devices to the platform, they only need to invoke AT commands, accelerating device interconnection and commissioning.
- The MCU does not need to integrate the MQTT protocol stack or IoT Device SDK Tiny, greatly reducing MCU resource consumption.
- Huawei releases certified modules on HUAWEI CLOUD Marketplace so that device manufacturers and service providers can purchase these certified modules to quickly connect to HUAWEI CLOUD IoT.

The following figure shows how a certificated module is used to connect a device to the platform.



Recommended Modules

Table 4-1 Certificated modules with pre-integrated Huawei SDKs

Module	Manufacturer	Model
4G Cat1 module	Fibocom	L610
	China Mobile IoT	ML302
4G Cat4 module	Quectel	EC20CEFASG
	Quectel	EC20CEHDLG
	Neoway	N720
NB-IoT module	China Mobile IoT	M5319-A

NOTE

- The LTE Cat4 module applies to the scenarios where the service data transmission rate ranges from 50 Mbit/s to 150 Mbit/s. The LTE Cat1 module applies to the scenarios where the service data transmission rate ranges from 5 Mbit/s to 10 Mbit/s.
- If you cannot find a required module in the preceding list, **submit a service ticket** to describe your service scenario and requirements.

Table 4-2 Modules that are not integrated with Huawei SDKs but have passed Huawei test

Module	Manufacturer	Model
NB-IoT module	Quectel	BC39
		BC95
		BC35
		BC26
		BC28
	Neoway	N27
		N25
		N21
	DWnet	TPB41
		TPB23
	Yuchen Technology	CFB-608
	Lierda	NB86-G
4G Cat4 module	Yuge	CLM920_NC5

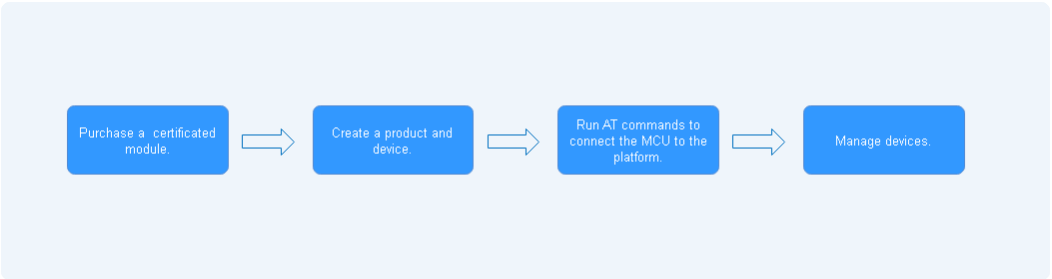
Module	Manufacturer	Model
		CLM920_NC3
	Quectel	EC20
4G Cat1 module	Neoway	N58
	Quectel	EC200S
2G/3G/4G module	Quectel	M25
ZigBee intelligent module	SHUNCOM	SZ05
5G module	Huawei	MH5000
LoRa module	Neoway	LR70
	WINEXT	M100C

Prerequisites

- The SIM card data service has been enabled, and the module can access the Internet.
- You have subscribed to the IoTDA service.

Development Process

The figure below shows the process for a manufacturer to develop a device.



- Purchase a HUAWEI CLOUD certificated module.
- Create a product and device on the IoTDA console.
- Run AT commands to connect the MCU to the HUAWEI CLOUD IoT platform and to receive data from and send data to the platform.
- Manage devices on the IoTDA console.

Purchasing a Certificated Module

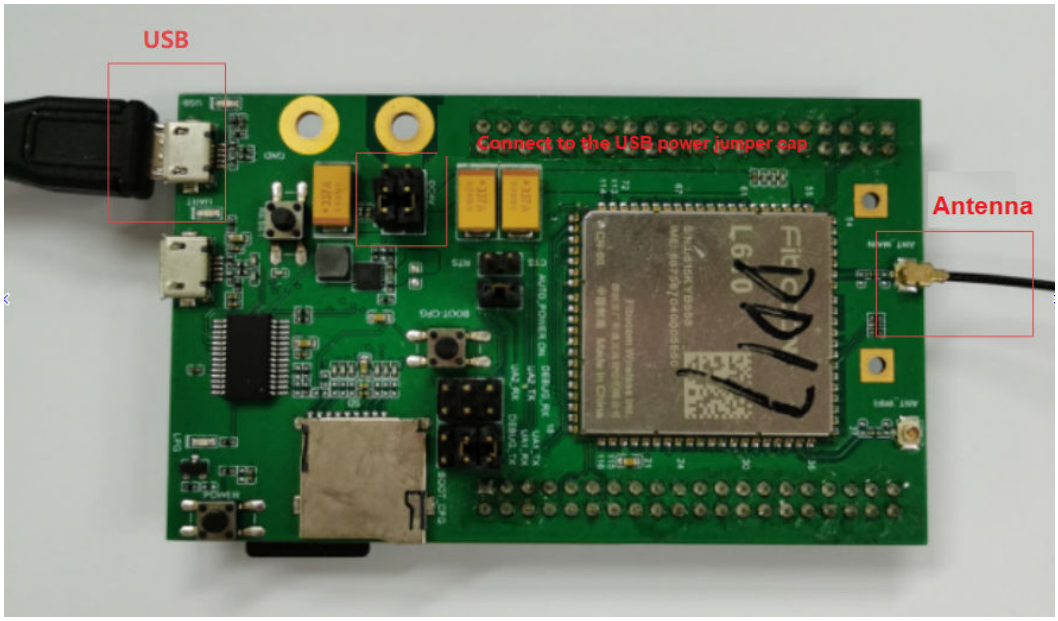
Step 1 Visit HUAWEI CLOUD Marketplace.

Step 2 Purchase the required module. For details on available modules, see **Table 4-1**.

----End

Connecting Hardware

Insert a 4G card into the SIM card slot. Ensure that the notch of the card faces inwards and the chip faces upwards. (This document uses the L610 module as an example.)



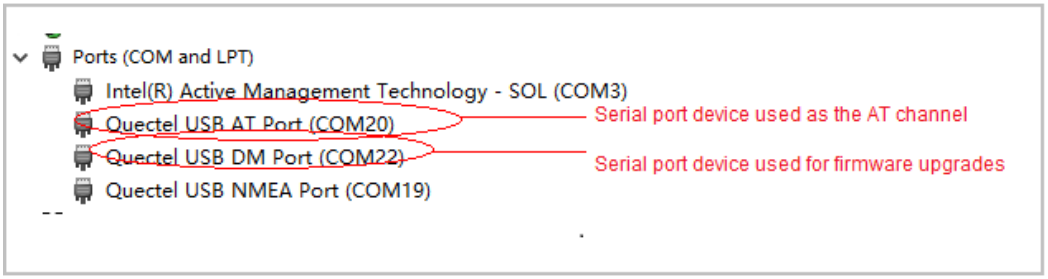
Installing the USB Driver

- Install the USB driver.
  a. Run the installation file and perform the installation as prompted.

  NOTE

  The USB driver version varies according to the device manufacturer. Contact the device manufacturer to obtain the required driver.

  b. After the driver is installed, connect the USB port of the development board to the PC and power on the PC. You can view the serial port devices in the device manager.

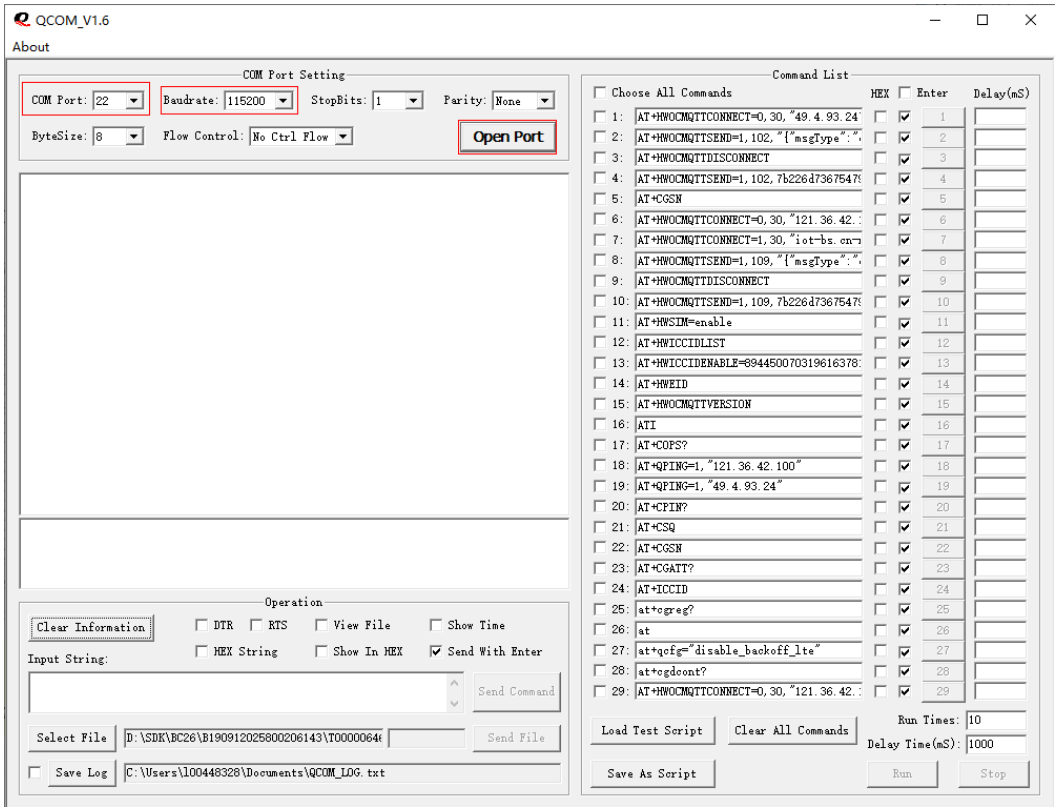


- Use a serial port tool to debug AT commands.
  a. Run the installation file and perform the installation as prompted.

  NOTE

  The version of the serial port tool varies according to the device manufacturer. Contact the device manufacturer to obtain a serial port tool that meets the requirements.

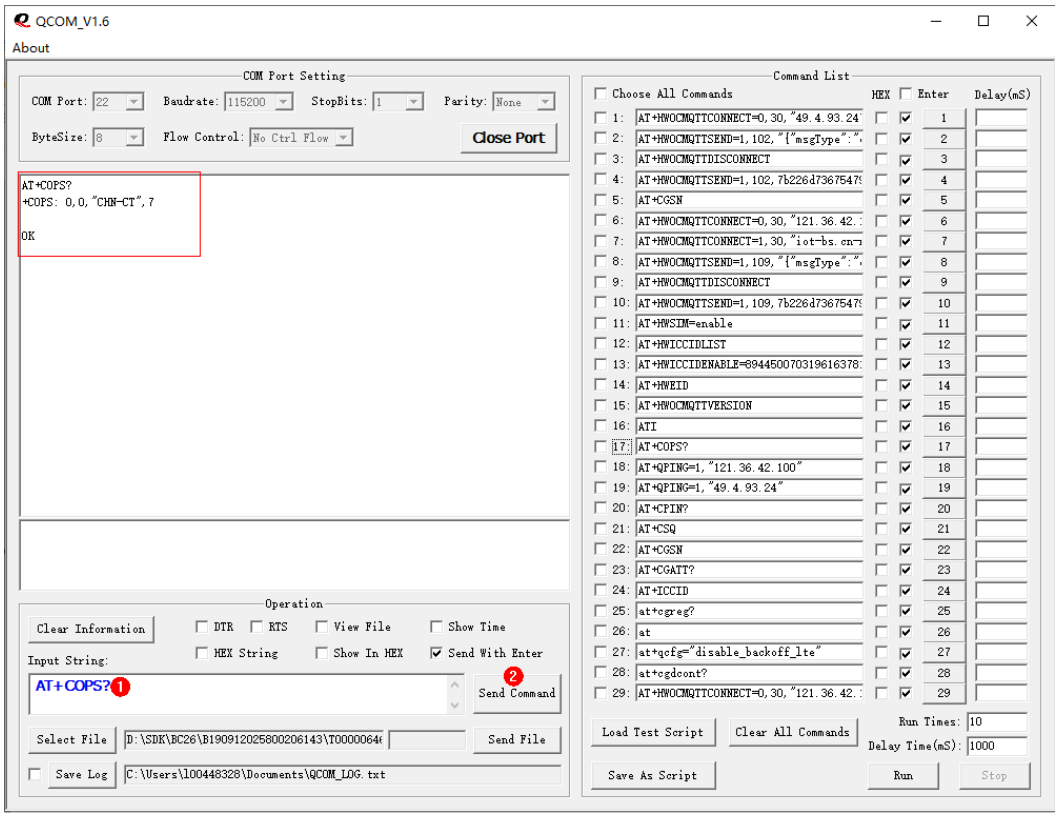
b. Open the serial port tool, select an AT serial port enumerated in **2**, set the baud rate to **115200**, and click **Open Port**.



NOTE

Ensure that the settings are correct. Otherwise, the AT command cannot be parsed or an error will occur during parsing.

c. Run the **AT+COPS?** command. Click **Send Command**. If **OK** is returned, the network registration is successful. Otherwise, check the settings and hardware cable connections.



NOTE

If the last digit of **+COPS: 0,0,"CHN-CT",7** in the returned message is not 7, the network is faulty. Replace the SIM card or check whether the SIM card can access the Internet.

Creating a Product and Device

Step 1 Create a product that uses MQTT by following the instructions provided in **Creating a Product**.

Step 2 **Register a device**.

NOTE

After the device is registered, keep the device ID and secret properly. The secret cannot be retrieved. If you forget the secret, click **Reset Secret** on the device details page to obtain a new one.

Step 3 Access the **IoTDA console** to obtain the MQTT/MQTTS device connection address. If MQTT is used, the port is 1883. If MQTTS is used, the port is 8883.

----End

Connecting to the Platform

The module provides AT commands in two encoding modes to connect to HUAWEI CLOUD: ASCII and hexstring. ASCII indicates the original encoding mode, and hexstring indicates the hexadecimal encoding mode.

- Using the ASCII mode

a. Connect to the platform. Send the command **AT +HMCON=bs,lifetime,"*serverip*","*serverport*","*deviceID*","*passwd*",*codec***, for example, **AT+HMCON=0,300,"iot-mqtts.cn-north-4.myhuaweicloud.com","8883","deviceID","passwd",0**. If **+HMCON OK** is received, the device is connected to the platform.

   The parameters in the preceding command are described as follows:

   - **bsmode**: whether device provisioning is used. Set this parameter to **0**. The value **0** means that the device is directly connected to the platform, and **1** means that the device is connected to the platform through device provisioning.
   - **lifetime**: MQTT heartbeat time. The default value is 300.
   - **serverip**: MQTT/MQTTS connection address. For details, see **Platform Connection Information**. If the device is connected to the platform through device provisioning, set this parameter to the address provisioned by the device. For details, see Device Provisioning.
   - **serverport**: port for device access. If MQTT is used, the port is 1883. If MQTTS is used, the port is 8883. If device provisioning is used, the service provisioning port is used.
   - **deviceID**: device ID returned for the registered device. For details, see 2.
   - **passwd**: secret set during device registration. For details on how to obtain the secret, see 2. If the certificate mode is used, you can leave this parameter unspecified. However, you must set the public key and private key certificates of the device in advance.
   - **codec**: data transmission mode. Set this parameter to **0** or **1**. The value **0** indicates the ASCII mode, and **1** indicates the hexstring mode. If the ASCII mode is used, the data mode is **len,ascii_payload**, for example, **2,"ab"**. If the hextring mode is used, the value is **2,"6162"**.

b. Subscribe to a custom topic. Send the **AT+HMSUB=*qos*,*topic*** command, for example, **AT+HMSUB=0,"$oc/devices/device_id/user/mytopic"**. If **+HMSUB OK** is received, the subscription is successful.

   The parameters in the preceding command are described as follows:

   - **qos**: QoS of the topic. The default value is **0**.

- **topic**: a new custom topic. For details, see **Adding a Custom Topic**. Set the device operation permission to **Subscribe** and replace **deviceID** with the actual device ID.

c. Report a message. Send the **AT+HMPUB=*qos*,*topic*,*payload_len*,*payload*** command, for example,

AT+HMPUB=0,"$oc/devices/device_id/user/mytopic",16,"{\"test\":\"hello\"}". If **+HMPUB OK** is received, the reporting is successful.

NOTE

The payload is in ASCII mode. The string must start and end with double quotation marks (""), and the special characters in the string must be escaped.

The parameters in the preceding command are described as follows:

- **qos**: QoS defined in MQTT. The recommended value is **0**.
- **topic**: a new custom topic. For details, see **Adding a Custom Topic**. Set the device operation permission to **Publish** and replace **deviceID** with the actual device ID.
- **payload_len**: length of the reported message, excluding the slash (\).
- **payload**: reported message.

d. Report a property. Send the **AT+HMPUB=*qos*,*topic*,*payload_len*,*payload*** command, for example,

AT+HMPUB=0,"$oc/devices/device_id/sys/properties/report",82,"{\"services\":[{\"service_id\":\"Clock\",\"properties\":{\"card_no\":\"3028\",\"use_type\":1}}]}". If **+HMPUB OK** is received, the reporting is successful. You can view the reported property values on the device details page.

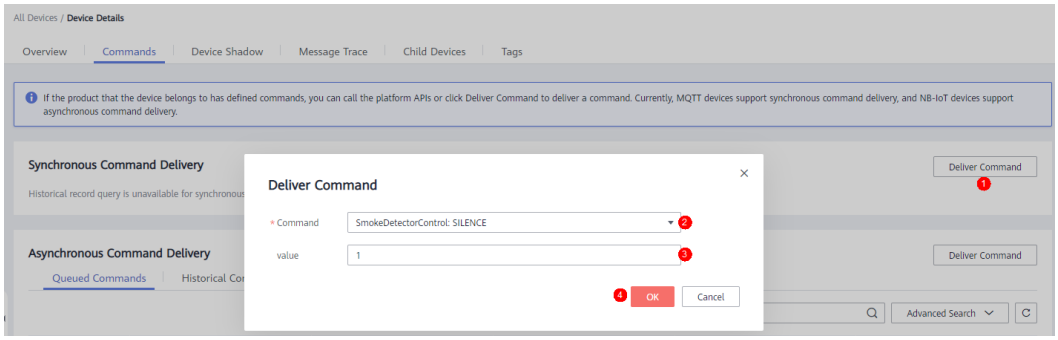
NOTE

Before reporting properties, customize a product model or use the preconfigured product model. For details, see **Developing a Product Model Online** and **Preconfigured Product Models**.

- **qos**: QoS defined in MQTT. The recommended value is **0**.
- **topic**: topic preconfigured on the platform. For more topics, see **Topic Definition**. Replace **deviceID** with the actual device ID.
- **payload_len**: length of the reported property, excluding the slash (\).
- **payload**: reported property.

e. Deliver a command. On the **Commands** tab page of the device details page of the IoTDA console, click **Deliver Command** on the right of **Synchronous Command Delivery**. Select the command to deliver and the command value. After the delivery is successful, the device receives **+HMREC:*topic*,*payload_len*,*payload***, for example, **+HMREC: "$oc/devices/device_id/sys/commands/request_id={request_id}{"paras":{"value":1},"service_id":"SmokeDetectorControl","command_name":"QUITSILENCE"}",86,{"paras":{"value":**

1},"service_id":"SmokeDetectorControl","command_name":"QUITSILENCE"}.



The parameters in the preceding command are described as follows:

- **qos**: QoS defined in MQTT. The recommended value is **0**.
- **topic**: topic preconfigured on the platform. For more topics, see **Topic Definition**. Replace **deviceID** with the actual device ID. **{request_id}** is used to uniquely identify the request. If this parameter is carried in a message sent by a device, ensure that the parameter value is unique on the device by using an incremental number or UUID. If this parameter is carried in a message received by a device, the parameter value needs to be also carried in the response message sent to the platform.
- **payload_len**: length of the delivered command, excluding the slash (\).
- **payload**: delivered command.

f. Unsubscribe from the custom topic. Send the **AT+HMUNS="*topic*"** command, for example, **AT+HMUNS="$oc/devices/deviceID/user/mytopic"**. If **+HMUNS OK** is received, the unsubscription is successful.

In the preceding command, topic is the custom topic added in **2**. Replace **deviceID** with the actual device ID.

g. Disconnect the device from the platform by sending the **AT+HMDIS** command.

h. Set the server or client certificate.

- To set a CA certificate, run **AT+HMPKS=type,para1,[para2],"Certificate"**, for example, **AT+HMPKS=0,1360**.
- To set a client certificate, run **AT+HMPKS=type,para1,[para2],"Certificate"**, for example, **AT+HMPKS=1,1022**.
- To set a private key certificate, run **AT+HMPKS=type,para1,[para2],"Certificate"**, for example, **AT+HMPKS=1,1732**.

NOTE

- **type**: The value can be **0**, **1**, or **2**, indicating a CA certificate, client certificate, and private key certificate, respectively. All the certificates are transmitted using **para1**. If a password is available, the password is transmitted through **para2**.
- **para1/[para2]**: The **para1** parameter specifies the certificate. If it is left blank, the certificate is to be deleted. The **para2** parameter specifies the password of the private certificate. It is valid only when the private certificate is set and the certificate is transmitted in the PEM format.
- **Certificate**: character length of the certificate content.

- Using the hexstring mode

a. Connect to the platform by sending the command **AT +HMCON=bs,lifetime,"*serverip*","*serverport*","*deviceID*","*passwd*",*codec***, for example, **AT+HMCON=0,300,"iot-mqtts.cn-north-4.myhuaweicloud.com","8883","deviceID","passwd",0**. If **+HMCON OK** is received, the device is connected to the platform.

   For details on the parameters, see **1**.

b. Subscribe to a custom topic by sending the **AT+HMSUB=qos,*topic*** command, for example, **AT+HMSUB=0,"$oc/devices/device_id/user/mytopic**. If **+HMSUB OK** is received, the subscription is successful.

   For details on the parameters, see **2**.

c. Report a message by sending the **AT +HMPUB=qos,*topic*,*payload_len*,*payload*** command, for example, **AT +HMPUB=0,"$oc/devices/device_id/user/mytopic", 16,7b2274657374223a2268656c6c6f227d**.

   Note: The payload is in hexadecimal data format and can be a hexadecimal character string without double quotation marks at the beginning or end.

   For details on the parameters, see **3**.

d. Deliver a command. On the **Commands** tab page of the device details page of the IoTDA console, click **Deliver Command** on the right of **Synchronous Command Delivery**. Select the command to deliver and the command value. After the command is delivered, the device receives the **+HMREC,*topic*,*payload_len*,*payload*** command, for example,

   +HMREC: "$oc/devices/device_id/sys/commands/request_id={request_id}", 102,7B227061726173223A7B2276616C7565223A22313233343536373839 3071617A77737865646372667674676279686E756A6D696B6F6C70227D2 C22736572766963655F6964223A224E42444F4F52222C22636F6D6D616E 645F6E616D65223A2273656E64227D

   For details on the parameters, see **5**.

e. Unsubscribe from the custom topic by sending the **AT+HMUNS="*topic*"** command, for example, **AT+HMUNS="$oc/devices/device_id/user/mytopic"**. If **+HMUNS OK** is received, the cancellation is successful.

   For details on the parameters, see **6**.

f. Disconnect the device from the platform by sending the **AT+HMDIS** command.

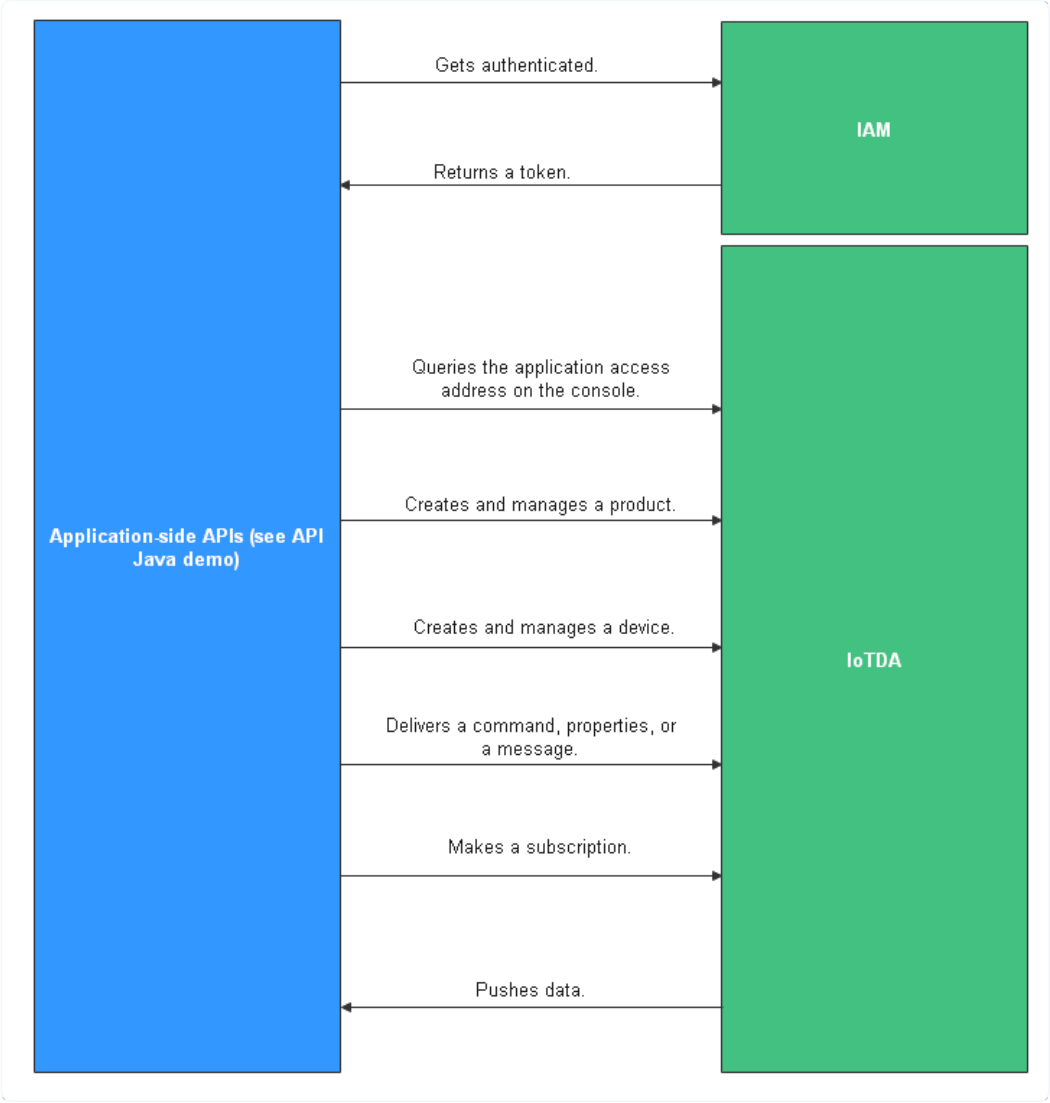
Device Management

The platform supports batch device management, **remote control and monitoring**, **OTA upgrades**, and flexible **data forwarding** to other HUAWEI CLOUD services.

5 Development on the Application Side

5.1 API

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 product, device, subscription, and rule management, as well as device command delivery.



1. An application must get authenticated by Identity and Access Management (IAM) and obtain a token. For details on how to obtain a token, see **Debugging the API Used to Obtain the Token for an IAM User**.
2. The application can implement functions such as product management, device management, command/property/message delivery, subscription, and push message receipt. For details on the functions, see the following description, as well as **API JAVA Demo** or **Debugging Using Postman**.

API Introduction

API Group	Scenario
Subscription Management	Applications subscribe to resources provided by the platform. If the subscribed resources change, the platform notifies the applications of the change.

API Group	Scenario
Tag Management	Applications bind tags to or unbind tags from resources. Currently, only devices support tags.
Batch Task	Applications perform batch operations on devices connected to the platform. • Software and firmware can be upgraded in batches, and devices can be created, deleted, frozen, or unfrozen in batches. • Up to 10 unfinished tasks of the same type is allowed for a single user. After the maximum number is reached, new tasks cannot be created.
Device CA Certificate Management	Applications manage device CA certificates, including uploading, verifying, and querying certificates. The platform supports device access authentication using certificates.
Device Group Management	Applications manage device groups, including managing device group information and devices in a device group.
Device Message	Applications transparently transmit messages to devices.
Product Management	Applications manage product models that have been imported to the platform. (A product model defines the capabilities or features of all devices under a product.)
Device Management	Applications manage basic device information and device data.
Device Shadow	Applications manage the device shadow, which 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 Command	Applications deliver commands defined in the product model to devices through the platform.
Device Property	Applications deliver properties defined in the product model to devices through the platform.

API Group	Scenario
Data forwarding and **Device Linkage**	Applications set rules to implement service linkage or forward data to other HUAWEI CLOUD services. Device linkage and data forwarding rules are available. • Device linkage: You can set trigger conditions and actions. When the preset triggering conditions are met, the corresponding actions are triggered, such as delivering commands, sending notifications, reporting alarms, and clearing alarms. • Data forwarding: You can set forwarding data, set forwarding targets, and start rules. 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.

5.2 Subscription and Push

5.2.1 Overview

A device can connect to and communicate with the platform. The device reports data to the platform using custom topics or product models. After the subscription/push configuration on the console is complete, the platform pushes messages about device lifecycle changes, reported device properties, reported device messages, device message status changes, device status changes, and batch task status changes to the application.

The platform supports two subscription modes: HTTP/HTTPS and AMQP.

- HTTP/HTTPS subscription/push: An application calls the platform APIs **Creating a Rule Trigger Condition**, **Creating a Rule Action**, and **Modifying a Rule Trigger Condition** to configure and activate rules. The platform pushes the changed device service details and management details to the application with a specified URL. (Service details include device lifecycle management, device data reporting, device message status, and device status. Management details include software/firmware upgrade status and result.)
- AMQP subscription/push: Data can be forwarded without interconnecting with other HUAWEI CLOUD services. An application calls the platform APIs **Creating a Rule Trigger Condition**, **Creating a Rule Action**, and **Modifying a Rule Trigger Condition** to configure and activate rules. After a connection is established between the AMQP client and the platform, the platform pushes the changes to a specified AMQP message queue based on the type of data subscribed. For details, see **5.2.3 AMQP Subscription/Push**.

Subscription/ Push	Application Scenario	Advantages and Disadvantages	Restrictions
HTTP/HTTPS subscription/ push	An application functions as the server and passively receives messages from the platform.	Data cannot be obtained proactively.	-
AMQP subscription/ push	An application functions as the client and proactively pulls messages from the platform or passively receives messages from the platform by means of listening.	Data can be obtained proactively.	For details, see **Connection Specifications**.

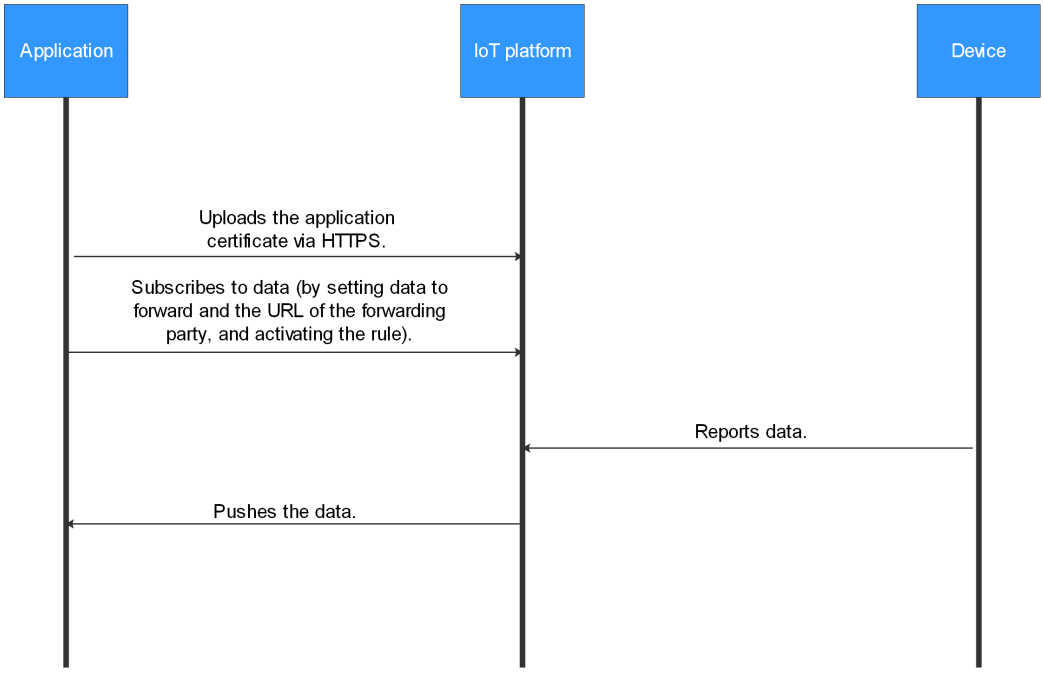
5.2.2 HTTP/HTTPS Subscription/Push

Overview

Subscription: An application calls the platform APIs **Creating a Rule Trigger Condition**, **Creating a Rule Action**, and **Modifying a Rule Trigger Condition** to configure and activate rules, in order to obtain changed device service details and management details. (Service details include device lifecycle management, device data reporting, device message status, and device status. Management details include software/firmware upgrade status and result.) The URL of the application, also called the callback URL, must be specified during subscription. **Click here to see what is a callback URL?**

Push: After a subscription is successful, the platform pushes the corresponding change to a specified callback URL based on the type of data subscribed. (For details on the pushed content, see **Transferring Data**.) If an application does not subscribe to a specific type of data notification, the platform does not push the data to the application even if the data has changed. The platform pushes data, in JSON format, using HTTP or HTTPS. HTTPS requires authentication and is more secure. Therefore, HTTPS is recommended.

The figure below shows the subscription and push process.



Before pushing HTTPS messages to an application, the platform must verify the application authenticity. Therefore, the application CA certificate must be loaded to the platform. (You can **use a commissioning certificate** during commissioning and replace it with a commercial certificate during commercial use to avoid security risks.)

Push mechanism: After receiving a push message from the platform, the application returns a 200 OK message. If the application does not respond within 15 seconds or returns a 501, 502, 503, or 504 message, the message delivery fails. The platform caches the message for 10 minutes. Then the platform retries to push the message to each failed application in polling mode. If the retry also fails and the message cache time elapses, the platform does not attempt delivery again. If the platform fails to send a push message 10 consecutive times within the message cache time, the platform sets the callback URL to invalid and checks the validity of all failed URLs in polling mode. If a URL is confirmed to be valid, the platform resets the URL to valid. You can log in to the IoTDA console, choose **Resource Spaces** in the navigation pane, click **View** in the row of a resource space, and view the URL status on the **Subscription/Push** tab page.

Subscribing to Data

After connecting to IoTDA, an application calls an API to subscribe to data.

- For details on how to configure HTTP or HTTPS subscriptions on the console, see **Configuring HTTP/HTTPS Subscription** and **Loading the CA Certificate**.
- For details on how to subscribe to data through APIs, see **Calling APIs**, **Creating a Rule Trigger Condition**, **Creating a Rule Action**, and **Modifying a Rule Trigger Condition**.

Format of Pushed Data

For details on the format of data pushed by the platform to applications after data subscription is created, see **Transferring Data**.

Loading the CA Certificate

If HTTPS is used, you must load the push certificate by following the instructions provided in this section. Then create a subscription task on the console by following the instructions provided in **Configuring HTTP/HTTPS Subscription**.

- If the application cancels the subscription and then re-subscribes the data again (with the URL unchanged), the CA certificate must be loaded to the platform again.
- If a subscription type (URL) is added, you must load the CA certificate corresponding to the URL to the platform. Even if the CA certificate used by the new URL is the same as that used by the original URL, the CA certificate must be loaded again.

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

Step 2 In the navigation pane, click **Resource Spaces**. On the page displayed, click **View** in the row of a resource space to access its details.

Step 3 On the **Subscription/Push** tab page, click **Configure Certificate**, set the parameters based on the data below, and click **OK** to load the certificate.

Parameter	Description
CA Certificate	A CA certificate from the application can be applied for and purchased in advance. **NOTE** You can **prepare a commissioning certificate** during commissioning. For security reasons, you are advised to replace the commissioning certificate with a commercial certificate during commercial use.
Domain/IP and Port	Specify the domain name or IP address and port used by the platform to push messages to the application. Set this parameter to the domain name or IP address and port in the URL of the API **Creating a Rule Action**, for example, **api.huawei.com:9001** and **172.0.1.2:8080**.
Check Common Name	Specify whether the common name of the CA certificate is to be verified to see whether the loaded certificate matches the applied certificate. It is recommended that the common name be verified.
Common Name	This parameter is displayed when **Check Common Name** is enabled. Obtain the name of the CA certificate from the certificate applicant.
SNI Support	If multiple servers use the same IP address and port, select **SNI Supported**, and set **Common Name** to the domain name of the server that is required to receive push messages. Then the specified server sends its device certificate to the platform. This parameter is not selected by default.

Parameter	Description
Use Device Certificate	Retain the default value **Disable**.

----End

Creating an X.509 Commissioning Certificate

A commissioning certificate, or a self-signed certificate, is used for authentication when the client accesses the server through HTTPS. When the platform uses HTTPS to push data to an application, the platform authenticates the application. This section uses the Windows operating system as an example to describe how to use OpenSSL to make a commissioning certificate. The generated certificate is in PEM format and the suffix is **.cer**.

The table below lists common certificate storage formats.

Storage Format	Description
DER	Binary code. The suffix is **.der**, **.cer**, or **.crt**.
PEM	Base64 code. The suffix is **.pem**, **.cer**, or **.crt**.
JKS	Java certificate storage format. The suffix is **.jks**.

NOTE

The commissioning certificate is used only for commissioning. During commercial use, you must apply for certificates from a trusted CA. Otherwise, security risks may occur.

Step 1 Visit **https://slproweb.com/products/Win32OpenSSL.html** to download and install OpenSSL.

Step 2 Open the CLI as user **admin**.

Step 3 Run **cd c:\openssl\bin** (replace **c:\openssl\bin** with the actual OpenSSL installation directory) to access the OpenSSL view.

Step 4 Generate the private key file **ca_private.key** of the CA root certificate.

```
openssl genrsa -passout pass:123456 -aes256  -out ca_private.key 2048
```

- **aes256**: encryption algorithm
- **passout pass**: private key password
- **2048**: key length

Step 5 Use the private key file of the CA root certificate to generate the file **ca.csr**.

```
openssl req -passin pass:123456 -new -key ca_private.key -out ca.csr -subj "/C=CN/ST=GD/L=SZ/O=Huawei/OU=IoT/CN=CA"
```

Modify the following information based on actual conditions:

- **C**: country, for example, CN
- **ST**: region, for example, GD
- **L**: city, for example, SZ
- **O**: organization, for example, Huawei
- **OU**: organization unit, for example, IoT
- **CN**: common name (the organization name of the CA), for example, CA

Step 6 Create the CA root certificate **ca.cer**.

```
openssl x509 -req -passin pass:123456 -in ca.csr -out ca.cer -signkey ca_private.key -CAcreateserial -days 3650
```

Modify the following information based on actual conditions:

- **passin pass**: The value must be the same as the private key password set in **4**.
- **days**: validity period of the certificate.

Step 7 Generate the private key file for the application.

```
openssl genrsa -passout pass:123456 -aes256 -out server_private.key 2048
```

Step 8 Generate the **.csr** file for the application.

```
openssl req -passin pass:123456 -new -key server_private.key -out server.csr -subj "/C=CN/ST=GD/L=SZ/O=Huawei/OU=IoT/CN=appserver.iot.com"
```

Modify the following information based on actual conditions:

- **C**: country, for example, CN
- **ST**: region, for example, GD
- **L**: city, for example, SZ
- **O**: organization, for example, Huawei
- **OU**: organization unit, for example, IoT
- **CN**: common name. Enter the domain name or IP address of the application.

Step 9 Use the CA private key file **ca_private.key** to sign the file **server.csr** and generate the server certificate file **server.cer**.

```
openssl x509 -req -passin pass:123456 -in server.csr -out server.cer -sha256 -CA ca.cer -CAkey ca_private.key -CAserial ca.srl -CAcreateserial -days 3650
```

Step 10 (Optional) If you need a **.crt** or **.pem** certificate, proceed this step. The following uses the conversion from **server.cer** to **server.crt** as an example. To convert the **ca.cer** certificate, replace **server** in the command with **ca**.

```
openssl x509 -inform PEM -in server.cer -out server.crt
```

Step 11 In the **bin** folder of the OpenSSL installation directory, obtain the CA certificate (**ca.cer**/**ca.crt**/**ca.pem**), application server certificate (**server.cer**/**server.crt**/**server.pem**), and private key file (**server_private.key**). The CA certificate is loaded to the platform, and the application server certificate and private key file are loaded to the application.

----End

Configuring HTTP/HTTPS Subscription

This section describes how to configure HTTP or HTTPS subscription on the console.

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

Step 2 In the navigation pane, choose **Rules** > **Data Forwarding**, and click **Create Rule** in the upper right corner.

Step 3 Set the parameters based on the table below and click **Create Rule**.

Parameter	Description
Rule Name	Specify the name of a rule to create.
Description	Describe the rule.
Data Source	• **Device**: Device information, such as device addition, deletion, and update, will be forwarded. When **Data Source** is set to **Device**, quick configuration is not supported. • **Device property**: A property value reported by a device in a resource space will be forwarded. Click **Quick Configuration** on the right and select the product, property, and service data to forward. • **Device message**: A message reported by a device in a resource space will be forwarded. Click **Quick Configuration** on the right and select data of a specified topic to forward. Select the product to which the topic belongs and enter the topic name. You can use a **custom topic** on the product details page or a **preset topic**. • **Device message status**: The status of device messages exchanged between the device and platform will be forwarded. For details on the device message status, see **Message Status**. When **Data Source** is set to **Device message status**, quick configuration is not supported. • **Device status**: The status change of a directly connected device in a resource space will be forwarded. Click **Quick Configuration** on the right to forward information about devices whose status is **Online**, **Offline**, or **Abnormal** to other services. For details on the status of devices directly connected to the IoT platform, see **Device Status**. • **Batch task**: The batch task status will be forwarded. When **Data Source** is set to **Batch Task**, quick configuration is not supported.
Trigger	After the data source is selected, the platform automatically matches the trigger event.
Resource Space	You can select a single resource space or all resource spaces. If **All resource spaces** is selected, quick configuration is not supported.

Step 4 Under **Set Forwarding Target**, click **Add**. On the displayed page, set the parameters based on the table below and click **OK**.

Parameter	Description
Forwarding Target	Select **Third-party application (HTTP push)**.
Push URL	Specify the domain name or IP address and port used by the platform to push messages to the application. for example, **api.huawei.com:9001** and **172.0.1.2:8080**. **NOTE** Ensure that the URL is the same as the domain name/IP address entered in **Loading the CA Certificate**.

Step 5 After the rule is defined, click **Start Rule** to start forwarding data to the HTTP or HTTPS message queue.

----End

FAQs

The following lists the frequently asked questions about the subscription and push service. For more questions, **click here**.

- **How Do I Obtain Certificates?**
- **How Do I Obtain the Callback URL When Calling the Subscription API?**
- **Can a Domain Name Be Used in a Callback URL?**
- **What Should I Do If an Error Code 503 Is Displayed?**
- **Why Does an Application Receive Multiple Push Messages After a Device Reports a Piece of Data?**
- **Why Is the Callback URL Invalid During the Subscription API Call?**
- **How Can I Obtain the subscriptionId Needed in Calling the API for Deleting a Subscription?**

APIs

Creating a Rule Action

Creating a Rule Trigger Condition

Modifying a Rule Trigger Condition

Forwarding Data

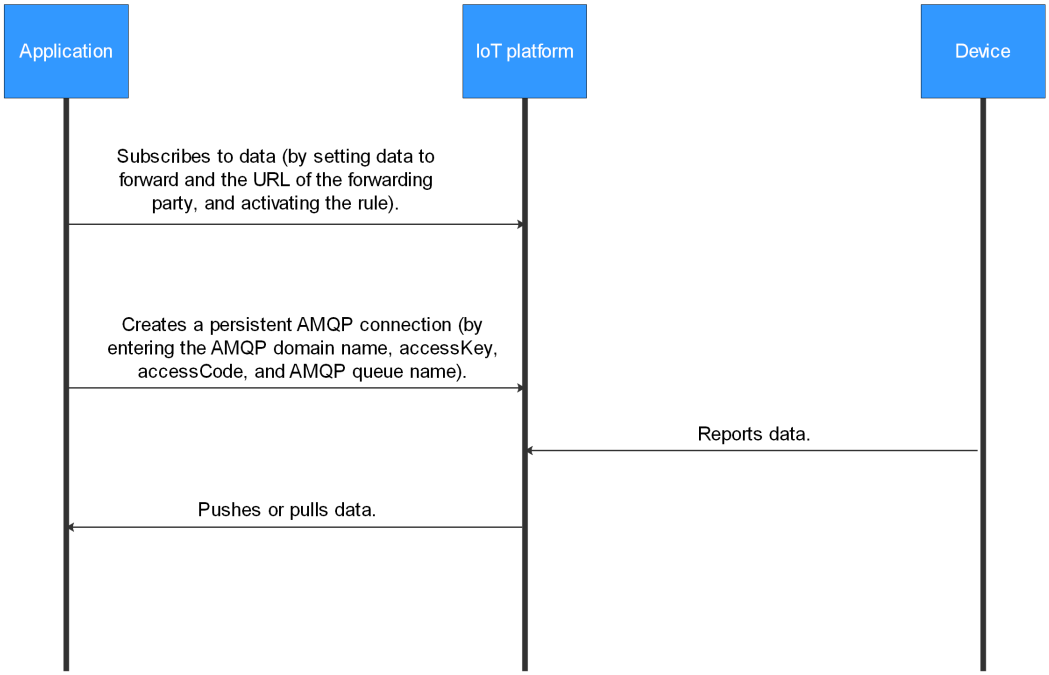
5.2.3 AMQP Subscription/Push

5.2.3.1 Overview

Subscription: AMQP is short for Advanced Message Queuing Protocol. You can create a subscription task on the IoTDA console. You can call platform APIs **Creating a Rule Trigger Condition**, **Creating a Rule Action**, and **Modifying a Rule Trigger Condition** to configure and activate rules for obtaining changed device service details and management details. (Service details include device

lifecycle management, device data reporting, device message status, and device status. Management details include software/firmware upgrade status and result.) The AMQP message channel must be specified during subscription creation.

Push: After a subscription is created, the platform pushes the corresponding change to the specified AMQP message queue based on the type of data subscribed. If an application does not subscribe to a specific type of data notification, the platform does not push the data to the application even if the data has changed. You can use the AMQP client to establish a connection with the platform to receive data. The figure below shows the subscription and push process.



Push mechanism: After receiving a message from the platform, the application returns a response. (The automatic response mode is recommended.) If the application does not pull data after the connection is established, data will be stacked on the server. When the maximum cache duration (one day) is reached, the platform clears the data. If the application does not respond in time after receiving the message and the persistent connection is interrupted, the corresponding data will be pushed again in the next connection established.

Subscribing to Data

After connecting to IoTDA, an application calls an API to subscribe to data.

- For details on how to configure subscriptions on the console, see **5.2.3.2 Configuring AMQP Server Subscription**.
- For details on how to subscribe to data through APIs, see **Calling APIs**, **Creating a Rule Trigger Condition**, **Creating a Rule Action**, and **Modifying a Rule Trigger Condition**.

Format of Pushed Data

For details on the format of data pushed by the platform to applications after data subscription is created, see **Transferring Data**.

APIs

Creating a Rule Action

Creating a Rule Trigger Condition

Modifying a Rule Trigger Condition

Transferring Data

Creating an AMQP Queue

Querying the AMQP List

Querying an AMQP Queue

Generating an Access Credential

5.2.3.2 Configuring AMQP Server Subscription

This topic describes how to set and manage AMQP server subscription on the IoT platform.

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

Step 2 In the navigation pane, choose **Rules** > **Data Forwarding**, and click **Create Rule** in the upper right corner.

Step 3 Set the parameters based on the table below and click **Create Rule**.

Parameter	Description
Rule Name	Specify the name of a rule to create.
Description	Describe the rule.

Parameter	Description
Data Source	• **Device**: Device information, such as device addition, deletion, and update, will be forwarded. When **Data Source** is set to **Device**, quick configuration is not supported. • **Device property**: A property value reported by a device in a resource space will be forwarded. Click **Quick Configuration** on the right and select the product, property, and service data to forward. • **Device message**: A message reported by a device in a resource space will be forwarded. Click **Quick Configuration** on the right and select data of a specified topic to forward. Select the product to which the topic belongs and enter the topic name. You can use a **custom topic** on the product details page or a **preset topic**. • **Device message status**: The status of device messages exchanged between the device and platform will be forwarded. For details on the device message status, see **Message Status**. When **Data Source** is set to **Device message status**, quick configuration is not supported. • **Device status**: The status change of a directly connected device in a resource space will be forwarded. Click **Quick Configuration** on the right to forward information about devices whose status is **Online**, **Offline**, or **Abnormal** to other services. For details on the status of devices directly connected to the IoT platform, see **Device Status**. • **Batch task**: The batch task status will be forwarded. When **Data Source** is set to **Batch Task**, quick configuration is not supported.
Trigger	After the data source is selected, the platform automatically matches the trigger event.
Resource Space	You can select a single resource space or all resource spaces. If **All resource spaces** is selected, quick configuration is not supported.

Step 4 Under **Set Forwarding Target**, click **Add**. On the displayed page, set the parameters based on the table below and click **OK**.

Parameter	Description
Forwarding Target	Select **AMQP message queue**.

Parameter	Description
Message Queue	Click **Select** to select a message queue. • If no message queue is available, create one. The queue name must be unique and can contain a maximum of 128 characters that consist of letters, numbers, underscores (_), hyphens (-), and vertical bars (|). Other characters such as the slash (/) are not allowed. • To delete a message queue, click **Delete** on the right of the message queue. **NOTE** A subscribed queue cannot be deleted.

Step 5 After the rule is defined, click **Enable Rule** to start forwarding data to the AMQP message queue.

----End

5.2.3.3 AMQP Client Access

After configuring and activating rules by calling the platform APIs **Creating a Rule Trigger Condition**, **Creating a Rule Action**, and **Modifying a Rule Trigger Condition**, connect the AMQP client to the IoT platform. Then run the AMQP client on your server to receive subscribed-to messages.

Protocol Version

For details on AMQP, see **AMQP**.

The IoT platform supports only AMQP 1.0.

Connection Establishment and Authentication

1. The AMQP client establishes a TCP connection with the platform and performs TLS handshake verification.

   NOTE

   To ensure security, the AMQP client must use TLS1.2 or a later version for encryption. Non-encrypted TCP transmission is not supported.

2. The client requests to set up a connection.
3. The client sends a request to the platform to establish a receiver link (a unidirectional channel for the platform to push data to the client).

   The receiver link must be set up within 15 seconds after the connection is set up on the client. Otherwise, the platform will close the connection.

   After the receiver link is set up, the client is connected to the platform.

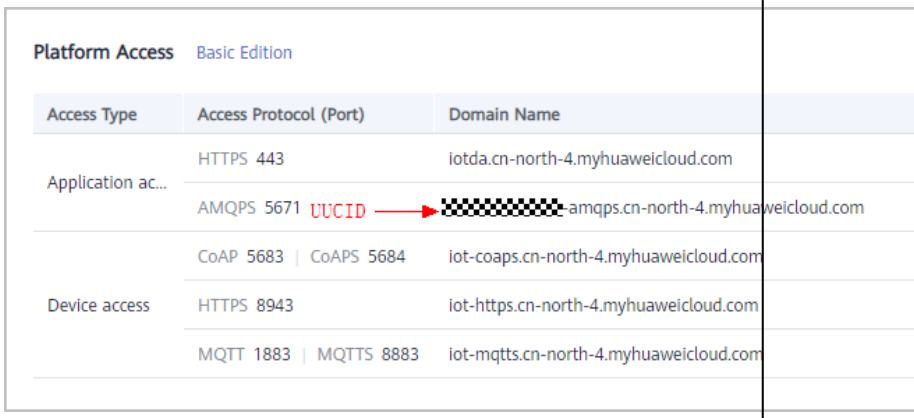
   NOTE

   Only one receiver link can be created for a connection, and sender links cannot be created. Therefore, the platform can push messages to the client, but the client cannot send messages to the platform.

Connection Configuration Parameters

The table below describes the connection address and connection authentication parameters for the AMQP client to connect to the platform.

- AMQP access domain name: amqps://${UUCID}.iot-amqps.cn-north-4.myhuaweicloud.com
- Connection string: amqps://${UUCID}.iot-amqps.cn-north-4.myhuaweicloud.com :5671?amqp.vhost=default&amqp.idleTimeout=8000&amqp.saslMechanisms=PLAIN

Parameter	Description
UUCID	Short for unique user connect ID, which is automatically generated for each account. You can view the UUCID on the **Overview** page of the **IoTDA console**. 
amqp.vhost	Currently, AMQP uses the default host. Only the default host is supported.
amqp.saslMechanisms	Connection authentication mode. Currently, PLAIN-SASL is supported.
idle-time-out	Heartbeat interval, in milliseconds. If the heartbeat interval expires and no frame is transmitted on the connection, the platform closes the connection.

- Port: 5671
- Client identity authentication parameters

  username = "accessKey=${accessKey}|timestamp=1599116822987|"

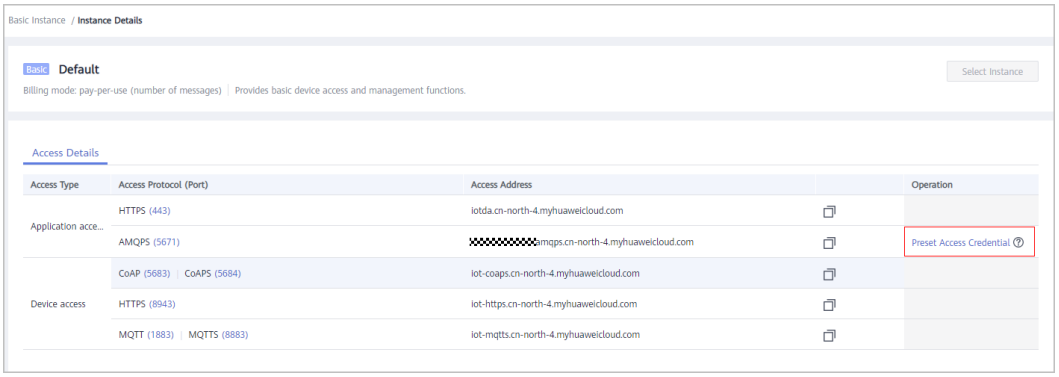
  password = "${accessCode}"

Parameter	Mandatory or Optional	Description
accessKey	Mandatory	An accessKey can be used to establish a maximum of 32 concurrent connections. When establishing a connection for the first time, preset the parameter by following the instructions provided in **Obtaining the AMQP Access Credential**.
timestamp	Mandatory	Indicates the current time. The value is a 13-digit timestamp, accurate to milliseconds. The server verifies the client timestamp. There is a 5-minute difference between the client timestamp and server timestamp.
accessCode	Mandatory	The value can contain a maximum of 256 characters. When establishing a connection for the first time, preset the parameter by following the instructions provided in **Resources**. If the accessCode is lost, you can call the API **Generating an Access Token** or follow the instructions provided in **Obtaining the AMQP Access Credential** to reset the accessCode.

Obtaining the AMQP Access Credential

If an application uses AMQP to access the platform for data transfer, preset an access credential. You can call the API **Generating an Access Credential** or use the console to preset an access credential. The procedure for using the console to generate an access credential is as follows:

1. In the navigation pane, click **IoTDA Instances**. On the page displayed, click **Details** under **Basic Edition** to access the details.
2. Click **Preset Access Credential** to preset the accessCode and accessKey.



NOTE

If you already have an access credential, the accessKey cannot be used after you preset the access credential again.

Connection Specifications

Key	Documentation
Maximum number of queues that can be connected to a connection	10
Maximum number of queues for a user	100
Maximum number of connections for a tenant	32
Maximum number of cached messages for an IoTDA instance	9,000
Maximum number of concurrent connections	1,000
Cache duration of a message (days)	1

Receiving Push Messages

After the receiver link between the client and platform is established, the client can proactively pull data or register a listener to enable the platform to push data. The proactive mode is recommended, because the client can pull data based on its own capability.

5.2.3.4 Java SDK Access Example

An AMQP-compliant JMS client connects to the IoT platform and receives subscribed messages from the platform.

Requirements for the Development Environment

JDK 1.8 or later has been installed.

Obtaining the Java SDK

The AMQP SDK is an open-source SDK. If you use Java, you are advised to use the Apache Qpid JMS client. Visit **Qpid JMS 0.50.0** to download the client and view the instructions for use.

Adding a Maven Dependency

```
<!-- amqp 1.0 qpid client -->
 <dependency>
   <groupId>org.apache.qpid</groupId>
   <artifactId>qpid-jms-client</artifactId>
   <version>0.50.0</version>
 </dependency>
```


Code Samples

You can click **here** to obtain the Java SDK access example. For details on the parameters involved in the demo, see **5.2.3.3 AMQP Client Access**.

```
package com.huawei.iot.amqp.jms;

import org.apache.qpid.jms.JmsConnection;
import org.apache.qpid.jms.JmsConnectionFactory;
import org.apache.qpid.jms.JmsConnectionListener;
import org.apache.qpid.jms.message.JmsInboundMessageDispatch;
import org.apache.qpid.jms.transports.TransportOptions;
import org.apache.qpid.jms.transports.TransportSupport;

import javax.jms.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.net.URI;
import java.util.Hashtable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class HwIotAmqpJavaClientDemo{
    // Asynchronous thread pool. You can adjust the parameters based on service features or use other asynchronous processing modes.
    private final static ExecutorService executorService = new ThreadPoolExecutor(Runtime.getRuntime().availableProcessors(),
            Runtime.getRuntime().availableProcessors() * 2, 60,
            TimeUnit.SECONDS, new LinkedBlockingQueue<>(5000));

    public static void main(String[] args) throws Exception{
        // accessKey for the access credential.
        String accessKey = "${yourAccessKey}";
        long timeStamp = System.currentTimeMillis();
        // Method to assemble userName. For details, see AMQP Client Access.
        String userName = "accessKey=" + accessKey + "|timestamp=" + timeStamp;
        // accessCode for the access credential.
        String password = "${yourAccessCode}";
        // Assemble the connection URL according to the qpid-jms specifications.
        String connectionUrl = "amqps://${UUCID}.iot-amqps.cn-north-4.myhuaweicloud.com:5671?amqp.vhost=default&amqp.idleTimeout=8000&amqp.saslMechanisms=PLAIN";
        Hashtable<String, String> hashtable = new Hashtable<>();
        hashtable.put("connectionfactory.HwConnectionURL", connectionUrl);
        // Queue name. You can use DefaultQueue.
        String queueName = "${yourQueue}";
        hashtable.put("queue.HwQueueName", queueName);
        hashtable.put(Context.INITIAL_CONTEXT_FACTORY, "org.apache.qpid.jms.jndi.JmsInitialContextFactory");
        Context context = new InitialContext(hashtable);
        JmsConnectionFactory cf = (JmsConnectionFactory) context.lookup("HwConnectionURL");
        // Multiple queues can be created for one connection. Match queue.HwQueueName with queue.HwQueueName.
        Destination queue = (Destination) context.lookup("HwQueueName");

        // Trust the server.
        TransportOptions to = new TransportOptions(); to.setTrustAll(true);
        cf.setSslContext(TransportSupport.createJdkSslContext(to));

        // Create a connection.
        Connection connection = cf.createConnection(userName, password);
        ((JmsConnection) connection).addConnectionListener(myJmsConnectionListener);
        // Create a session.
        // Session.CLIENT_ACKNOWLEDGE: After receiving a message, manually call message.acknowledge().
        // Session.AUTO_ACKNOWLEDGE: The SDK automatically responds with an ACK message. (recommended processing)
        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        connection.start();
```



```
        // Create a receiver link.
        MessageConsumer consumer = session.createConsumer(queue);
        // Messages can be processed in either of the following ways:
        // 1. Proactively pull data (recommended processing). For details, see receiveMessage(consumer).
        // 2. Add a listener. For details, see consumer.setMessageListener(messageListener). The server
proactively pushes data to the client at an acceptable data rate.
        receiveMessage(consumer);
        // consumer.setMessageListener(messageListener);
    }

    private static void receiveMessage(MessageConsumer consumer) throws JMSException{
        while (true){
            try{
                // It is recommended that received messages be processed asynchronously. Ensure that the
receiveMessage function does not contain time-consuming logic.
                Message message = consumer.receive(); processMessage(message);
            } catch (Exception e) {
                System.out.println("receiveMessage hand an exception: " + e.getMessage());
                e.printStackTrace();
            }
        }

    }

    private static MessageListener messageListener = new MessageListener(){
        @Override
        public void onMessage(Message message){
            try {
                // It is recommended that received messages be processed asynchronously. Ensure that the
onMessage function does not contain time-consuming logic.
                // If the service processing takes a long time and blocks the thread, the normal callback after the
SDK receives the message may be affected.
                executorService.submit(() -> processMessage(message));
            } catch (Exception e){
                System.out.println("submit task occurs exception: " + e.getMessage());
                e.printStackTrace();
            }
        }
    };

    /**
     * Service logic for processing the received messages
     */
    private static void processMessage(Message message) {
        try {
            String body = message.getBody(String.class); String content = new String(body);
            System.out.println("receive an message, the content is " + content);
        } catch (Exception e){
            System.out.println("processMessage occurs error: " + e.getMessage());
            e.printStackTrace();
        }
    }

    private static JmsConnectionListener myJmsConnectionListener = new JmsConnectionListener(){
        /**
         * Connection established.
         */
        @Override
        public void onConnectionEstablished(URI remoteURI){
            System.out.println("onConnectionEstablished, remoteUri:" + remoteURI);
        }

        /**
         * The connection fails after the maximum number of retries is reached.
         */
        @Override
        public void onConnectionFailure(Throwable error){
            System.out.println("onConnectionFailure, " + error.getMessage());
        }
```



```
    /**
     * Connection interrupted.
     */
    @Override
    public void onConnectionInterrupted(URI remoteURI){
        System.out.println("onConnectionInterrupted, remoteUri:" + remoteURI);
    }

    /**
     * Automatic reconnection.
     */
    @Override
    public void onConnectionRestored(URI remoteURI){
        System.out.println("onConnectionRestored, remoteUri:" + remoteURI);
    }

    @Override
    public void onInboundMessage(JmsInboundMessageDispatch envelope){
        System.out.println("onInboundMessage, " + envelope);
    }

    @Override
    public void onSessionClosed(Session session, Throwable cause){
        System.out.println("onSessionClosed, session=" + session + ", cause =" + cause);
    }

    @Override
    public void onConsumerClosed(MessageConsumer consumer, Throwable cause){
        System.out.println("MessageConsumer, consumer=" + consumer + ", cause =" + cause);
    }

    @Override
    public void onProducerClosed(MessageProducer producer, Throwable cause){
        System.out.println("MessageProducer, producer=" + producer + ", cause =" + cause);
    }
  };
}
```

5.2.3.5 Node.js SDK Access Example

This topic describes how to use a Node.js AMQP SDK to connect to the HUAWEI CLOUD IoT platform and receive subscribed messages from the platform.

Development Environment

Node.js 8.0.0 or later is used.

Downloading the SDK

For the AMQP SDK using Node.js, rhea is recommended. Visit **rhea** to download the repository and view the user guide.

Adding Dependencies

Add the following dependencies to the **package.json** file:

```
"dependencies": {
   "rhea": "^1.0.12"
 }
```


Sample Code

You can click **here** to obtain the SDK access example. For details on the parameters involved in the demo, see **5.2.3.3 AMQP Client Access**.

```
const container = require('rhea');
// Obtain the timestamp.
var timestamp = Math.round(new Date() / 1000);

// Set up a connection.
var connection = container.connect({
    // Access domain name. For details, see AMQP Client Access.
    'host': '${UUCID}.iot-amqps.cn-north-4.myhuaweicloud.com',
    'port': 5671,
    'transport': 'tls',
    'reconnect': true,
    'idle_time_out': 8000,
    // Method to assemble username. For details, see AMQP Client Access.
    'username': 'accessKey=${yourAccessKey}|timestamp=' + timestamp + '|',
    // accessCode. For details, see AMQP Client Access.
    'password': '${yourAccessCode}',
    'saslMechannisms': 'PLAIN',
    'rejectUnauthorized': false,
    'hostname': 'default',
});

// Create a Receiver connection. You can use DefaultQueue.
var receiver = connection.open_receiver('${yourQueue}');

// Callback function for receiving messages pushed from the cloud
container.on('message', function (context) {
    var msg = context.message;
    var content = msg.body;
    console.log(content);
    // Send an ACK message. Note that the callback function should not contain time-consuming logic.
    context.delivery.accept();
 });
```

5.3 Java Demo

This topic describes how to use the sample code (Java) for calling APIs. For details on these APIs, see **API Reference on the Application Side**.

(Optional) Preparing the Java Development Environment

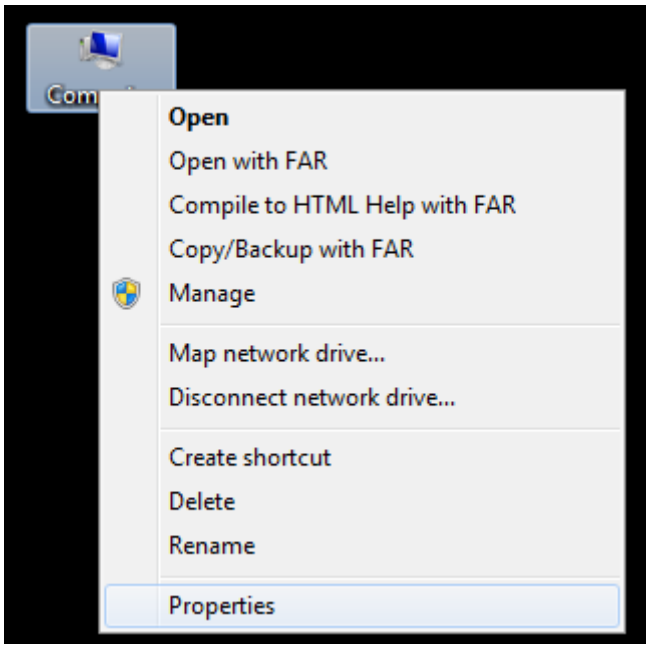
If you have prepared the Java development environment, skip this section.

This section describes how to install the JDK 1.8 and Eclipse in the Windows operating system. If you use another development environment, deploy the two tools based on project situations.

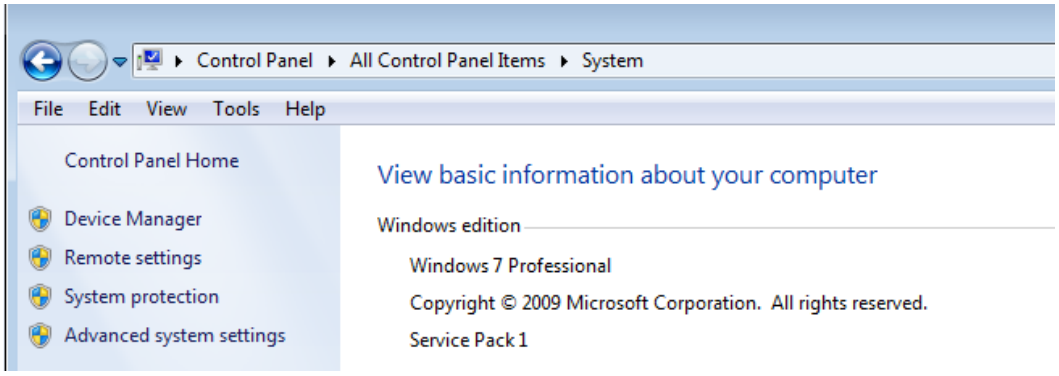
Step 1 Download JDK 1.8 (for example, **jdk-8u161-windows-x64.exe**) from the **Java JDK website**, and double-click it to install it.

Step 2 Configure Java environment variables.

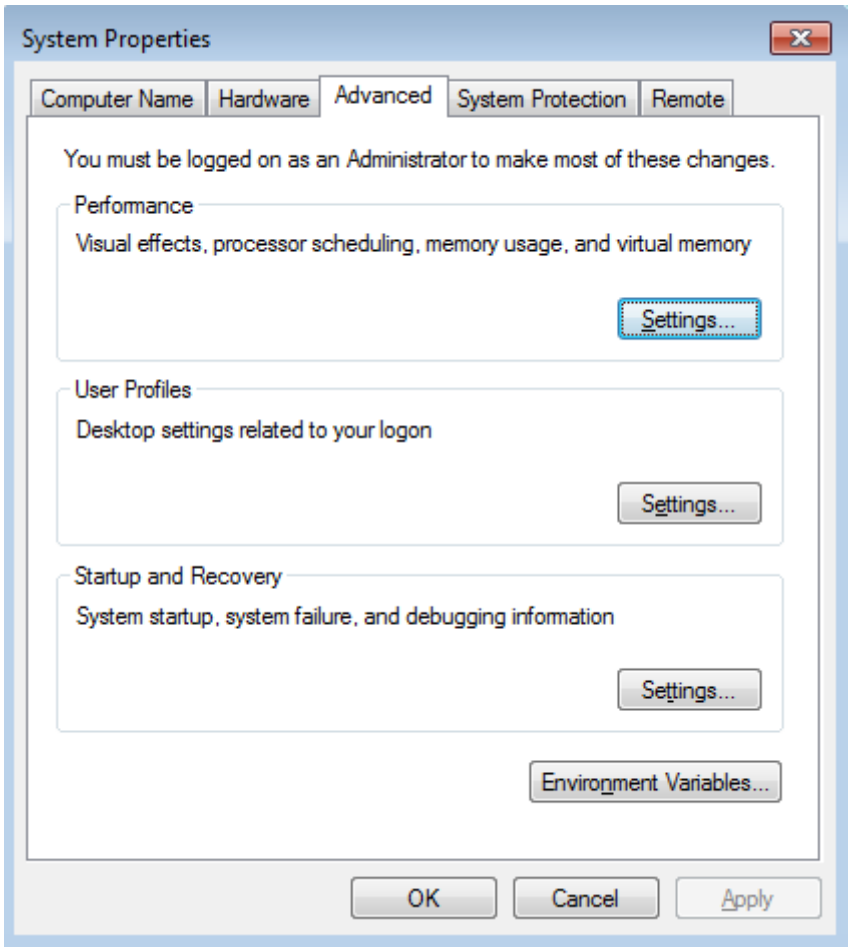
1. Right-click **Computer** and choose **Properties**.



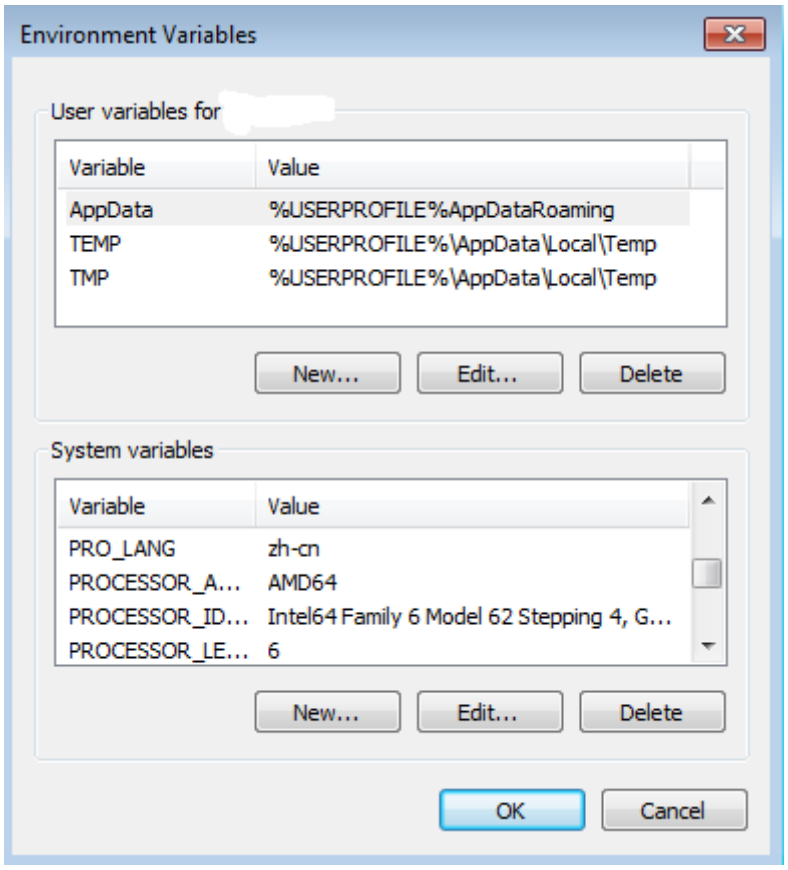
2. Select **Advanced system settings**.



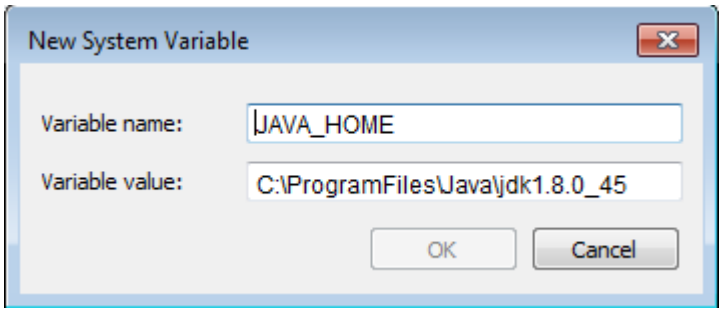
3. In the **System Properties** dialog box, choose **Advanced** > **Environment Variables**.



4. Configure the system variables. Configure the following three variables: **JAVA_HOME**, **Path**, and **CLASSPATH** (where the variable names are case-insensitive). If a variable name already exits, click **Edit**. If a variable name does not exist, click **New** to create one. Generally, the **Path** variable exists, and the **JAVA_HOME** and **CLASSPATH** variables need to be added.

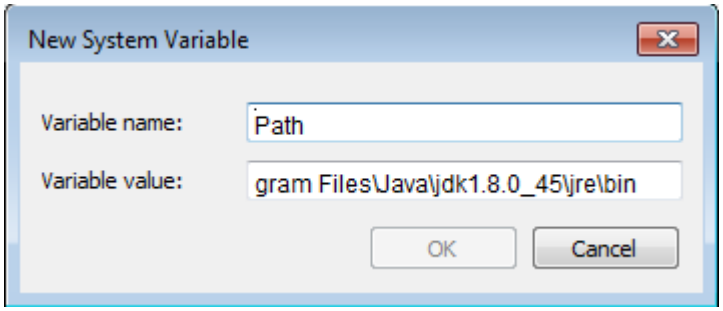


JAVA_HOME indicates the JDK installation path and is set to **C:\ProgramFiles\Java\jdk1.8.0_45**. This path contains the **lib** and **bin** files.



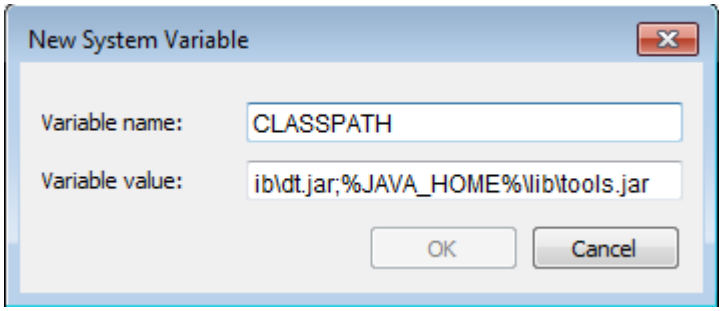
Path enables the system to recognize a Java command in any path. If the **Path** variable exists, add a path at the end of the variable value. Configuration example: **;C:\Program Files\Java\jdk1.8.0_45\bin;C:\Program Files\Java\jdk1.8.0_45\jre\bin**

Separate two paths using a semicolon (;).



CLASSPATH specifies the path of loaded Java classes (class or lib). Java commands can be identified only if they are contained in the class path. Configuration example: **.;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar**

Note: The path starts with a dot (.), indicating the current path.



5. Choose **Start** > **Run**, enter **cmd**, and run the following commands: **Java -version**, **java**, and **javac**. If the commands can be run, the environment variables are set.

```
C:\Users\z00293999>java -version
java version "1.8.0_45"
Java(TM) SE Runtime Environment (build 1.8.0_45-b15)
Java HotSpot(TM) Client VM (build 25.45-b02, mixed mode)
```

Step 3 Download the Eclipse installation package from the **IDEA website** and decompress it to the local directory.

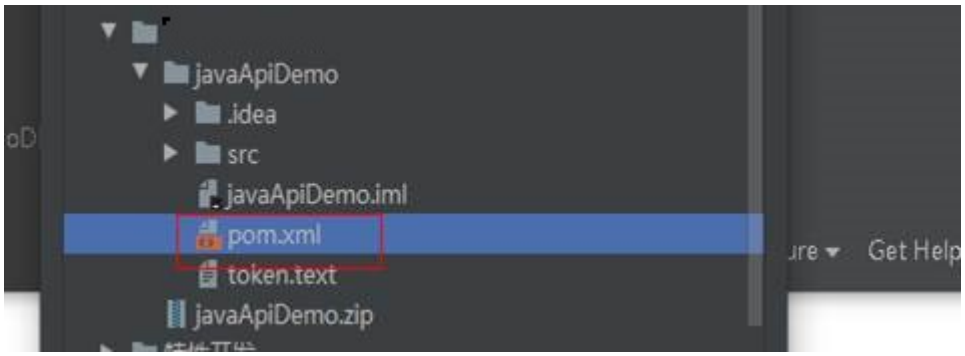
----End

Importing the Demo Project

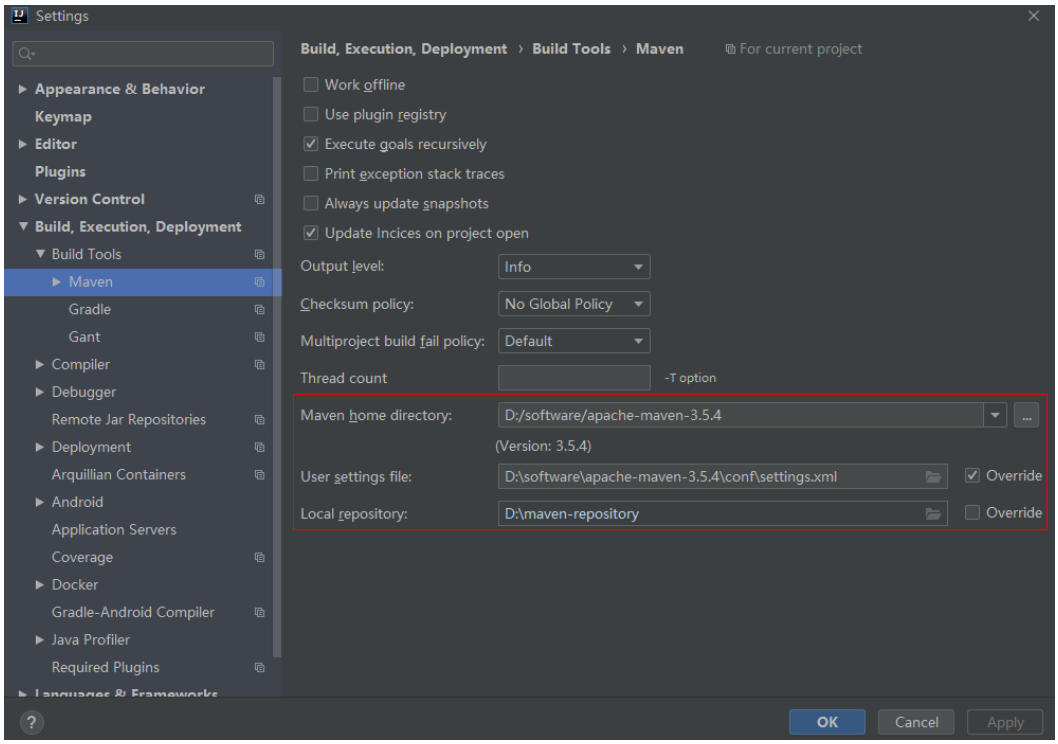
This section describes how to call APIs based on the Java sample code. Do not use the sample code for commercial use. For details on these APIs, see **API Reference on the Application Side**.

Step 1 **Download** and decompress the API demo in Java.

Step 2 Open IDEA, click **Import Project**, select **pom.xml** in the decompressed **demo** folder, and click **OK**.



Step 3 Choose **File** > **Setting** > **Build, Execution, Deployment** > **Build Tools** > **Maven**, set **User setting file** to the path of the **settings.xml** file of Maven, and set **Local repository** to the path of the local Maven repository.



----End

Obtaining a Token

Before accessing 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 the authentication token **X-Subject-Token** to the application.

This section describes how to call the authentication API based on the Java code sample of the API.

Step 1 In IDEA, choose **JavaApiDemo** > **src** > **main** > **java** > **com.huawei,.util** > **Constants.java**, and then change the values of **TOKEN_BASE_URL** and **IOTDM_BASE_URL**.

```
public class Constants {

    public static final String TOKEN_BASE_URL = "https://iam.cn-north-4.myhuaweicloud.com";
    public static final String IOTDM_BASE_URL = "https://iotda.cn-north-4.myhuaweicloud.com";

    public static final String TOKEN_ACCESS_URL = TOKEN_BASE_URL + "/v3/auth/tokens";

    public static final String DEVICE_COMMAND_URL = IOTDM_BASE_URL + "/v5/iot/%s/devices";
    public static final String PRODUCT_COMMAND_URL = IOTDM_BASE_URL + "/v5/iot/%s/products";
}
```

Parameters are described as follows:

- **TOKEN_BASE_URL**: Enter the address for interconnecting with IAM, that is, the IAM endpoint, which can be obtained from **IAM Regions and Endpoints**.
- **IOTDM_BASE_URL**: Enter the address for interconnecting with IoTDA, that is, the IoTDA endpoint, which can be obtained from **IoTDA Regions and Endpoints**.

  NOTE

  The endpoints vary depending on the region. Obtain the endpoints based on project conditions. For example, if you have subscribed to IoTDA in CN North-Beijing 4, obtain the endpoint of CN North-Beijing 4 from **IoTDA Regions and Endpoints**.

Step 2 In the imported sample code, choose **JavaApiDemo** > **src** > **main** > **java** > **com.huawei.demo.auth** > **Authentication.java**.

Change the account information to your own account information, right-click **Authentication.java**, and choose **Run Authentication.main()** to run the code.

```
AccessTokenDTO accessTokenDTO = new AccessTokenDTO();
AuthDTO authDTO = new AuthDTO();
DomainDTO domainDTO = new DomainDTO();
IdentityDTO identityDTO = new IdentityDTO();
PasswordDTO passwordDTO = new PasswordDTO();
UserDTO userDTO = new UserDTO();
ProjectDTO projectDTO = new ProjectDTO();
ScopeDTO scopeDTO = new ScopeDTO();
projectDTO.setName("cn-north-4");
scopeDTO.setProject(projectDTO);
domainDTO.setName("hwstaff_******");

userDTO.setName("hwstaff_********");
userDTO.setPassword("********");
userDTO.setDomain(domainDTO);

passwordDTO.setUser(userDTO);
List<String> method = new ArrayList<~>();
method.add("password");
identityDTO.setMethods(method);
identityDTO.setPassword(passwordDTO);
```

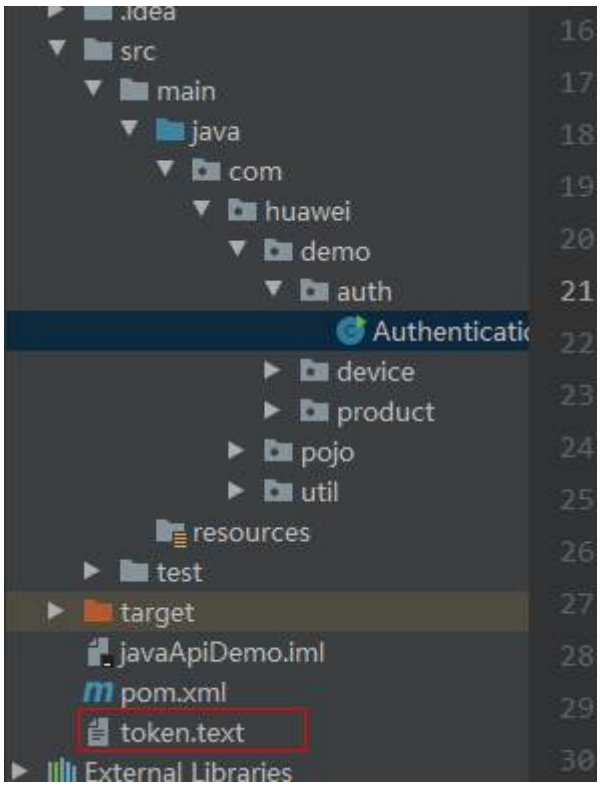
Step 3 View the response log on the console. If a token is obtained, the authentication is successful.

Keep the token secure. It will be used when you call other APIs.

```
D:\develop\java\bin\java.exe ...
MIIbZAYJKoZIhvcNAQcCoIIbVTCCG1ECAQExDTALBglghkgBZQMEAgEwghl2

Process finished with exit code 0
```

If no correct response is obtained, check whether the global constants are modified correctly or whether a network fault occurs.

Note: For each attempt to obtain a new token, the system preferentially retrieves the existing token stored in the file. If the token has expired, the system deletes the **token.text** file and obtains a new one.



----End

Device Registration (Token Authentication)

Before connecting a device to the platform, an application must call the API **Creating a Device**. Each device connecting to the platform carries the device ID to complete access authentication. For details, see **API Reference**.

This section describes how to call the API based on the Java sample code of the API.

Step 1 In IDEA, choose **JavaApiDemo** > **src** > **main** > **java** > **com.huawei.demo.device** > **CreateDevice.java**.

Modify parameters such as **nodeId**, **timeout**, **secret**, **deviceName**, and **productId**. For details on the parameter description, see the API **Creating a Device**.

Add the obtained token to the X-Auth-Token request header.


```
authInfo.setAuth_type("SECRET");
authInfo.setSecret("123456678");
authInfo.setSecure_access(true);
authInfo.setTimeout(300);

addDevice.setAuth_info(authInfo);
addDevice.setDescription("test device");
addDevice.setDevice_name("test_deviceName2");
addDevice.setNode_id("1111222223333444");
addDevice.setProduct_id("**********");

Map<String, String> headers = new HashMap<~>();
headers.put("Content-Type", "application/json");
headers.put("X-Auth-Token", token);
```

Step 2 In IDEA, right-click **CreateDevice.java** and choose **Run CreateDevice.main()** to run the code.

Step 3 View the response log on the console. If all types of subscriptions obtain the response "201" as well as **deviceId**, the subscription is successful.

```
D:\develop\java\bin\java.exe ...
HTTP/1.1 201
{"app_id":"58f68008ce5a4ec8b5caf3d1910ca69a","device_id":"5e5880f0f92c9902fc1e09a8_111122222333344455","node_id":"111122222333344455","gateway_id"
```

----End

Device Query (Token Authentication)

Applications can call the API **Querying a Device** to query details about a device registered with the platform.

This section describes how to call the API based on the Java code sample of the API.

Step 1 In IDEA, choose **JavaApiDemo** > **src** > **main** > **java** > **com.huawei.demo.device** > **QueryDeviceList.java**, and then modify the corresponding parameters.


```
String project_id = "23123";
String url = Constants.DEVICE_COMMAND_URL;
url = String.format(url, project_id);
Map<String, String> header = new HashMap<~>();
header.put("Content-Type", "application/json");
header.put("X-Auth-Token", token);

Map<String,String> map = new HashMap<String, String>();
map.put("device_name", "test_deviceName222");

HttpUtils httpUtils = new HttpUtils();
httpUtils.initClient();
StreamClosedHttpResponse httpResponse = httpUtils.doGet(url, header, map);
System.out.println(httpResponse.getStatusLine());
System.out.println(httpResponse.getContent());
```

Step 2 Right-click **QueryDeviceList** and choose **Run QueryDeviceList.main()** to run the code.

Step 3 View the response log on the console. If **deviceId** is obtained, the query is successful.

```
D:\develop\java\bin\java.exe ...
HTTP/1.1 200
{"devices":[{"app_id":"58f68008ce5a4ec8b5caf3d1910ca69a","device_id":"5e5880f0f92c9902fc1e09a8_111122222333344455","node_id":"111122222333344455","gateway_i
```

----End

Device Registration (AK/SK Authentication)

In addition to token authentication, AK/SK authentication is supported for calling platform APIs. This section describes how to call the AK/SK authentication API based on the sample code (Java) for calling APIs.

Step 1 In IDEA, choose **JavaApiDemo** > **src** > **main** > **java** > **com.huawei.demo.device** > **CreateDeviceByAK.java**, modify the corresponding parameters, and call the **SignUtil.signRequest()** method to sign the request.

```
addDevice.setAuth_info(authInfo);
addDevice.setDescription("test device");
addDevice.setDevice_name("test_deviceName2");
addDevice.setNode_id("1111222223333444");
addDevice.setProduct_id("5e09f371334dd4f337056da0");

Map<String, String> headers = new HashMap<~>();
headers.put("Content-Type", "application/json");

String project_id = "11111";
String url = Constants.DEVICE_COMMAND_URL;
url = String.format(url, project_id);

HttpRequestBase httpRequestBase = SignUtil.signRequest(url,  method: "POST",  headers,JsonUtils.Obj2String(addDevice),  params: null);

HttpUtils httpUtils = new HttpUtils();
httpUtils.initClient();

StreamClosedHttpResponse httpResponse = (StreamClosedHttpResponse)httpUtils.execute(httpRequestBase);
```

Called URL
Request method
Request header
Request message body
Parameters to be assembled in the URL

Step 2 In IDEA, choose **JavaApiDemo** > **src** > **main** > **java** > **com.huawei.demo.apig** > **SignUtil.java**, and modify the AK/SK in the **signRequest()** method. For details, see **Obtaining an AK/SK**.


```
public static HttpRequestBase signRequest(String url, String method, Map<String, S
    try {
        Request request = new Request();
        request.setKey("*********");
        request.setSecret("*********");
        if (body != null && body.length() > 0) {
            request.setBody(body);
        }
        if (params != null && !params.isEmpty()) {
            HttpUtils httpUtils = new HttpUtils();
            url = httpUtils.constructUri(url, params);
        }
        request.setUrl(url);
        for (String key : header.keySet()) {
            request.addHeader(key, header.get(key));
        }
```

Step 3 In IDEA, right-click **CreateDeviceByAK.java** and choose **Run CreateDeviceByAK.main()** to run the code.

Step 4 View the response log on the console. If all types of subscriptions obtain the response "201" as well as **deviceId**, the subscription is successful.

```
D:\develop\java\bin\java.exe ...
HTTP/1.1 201
{"app_id":"58f68008ce5a4ec8b5caf3d1910ca69a","device_id":"5e5880f0f92c9902fc1e09a8_111122222333344455","node_id":"111122222333344455","gateway_id"
```

----End

Device Query (AK/SK Authentication)

Applications can call the API **Querying a Device** to query details about a device registered with the platform.

This section describes how to call the API based on the Java code sample of the API.

Step 1 In IDEA, choose **JavaApiDemo** > **src** > **main** > **java** > **com.huawei.demo.device** > **QueryDeviceListByAK.java**, modify the corresponding parameters, sign the request, and replace the AK/SK in the signature method. For details, see **Obtaining an AK/SK**.


```
public static void main(String[] args) throws NoSuchAlgorithmException, KeyManagementException {
    String project_id = "23123";
    String url = Constants.DEVICE_COMMAND_URL;
    url = String.format(url, project_id);
    Map<String, String> header = new HashMap<~>();
    header.put("Content-Type", "application/json");

    Map<String, String> params = new HashMap<~>();
    params.put("node_id", "gaoshang_001");

    HttpRequestBase httpRequestBase = SignUtil.signRequest(url, method: "GET", header, body: null, params);

    HttpUtils httpUtils = new HttpUtils();
    httpUtils.initClient();

    StreamClosedHttpResponse httpResponse = (StreamClosedHttpResponse)httpUtils.execute(httpRequestBase);
    System.out.println(httpResponse.getStatusLine());
    System.out.println(httpResponse.getContent());
}
```

Request URL | Request method | Request header | Request message body | Parameters to be assembled in the URL

Step 2 Right-click **QueryDeviceListByAK** and choose **Run QueryDeviceListByAK.main()** to run the code.

Step 3 View the response log on the console. If **deviceId** is obtained, the query is successful.

```
D:\develop\java\bin\java.exe ...
HTTP/1.1 200
{"devices":[{"app_id":"58f68008ce5a4ec8b5caf3d1910ca69a","device_id":"5e5880f0f92c9902fc1e09a8_111122222333344455","node_id":"111122222333344455","gateway_i
```

----End

Development of Other APIs

For details on how to develop other APIs, see **API Reference**.

Performing Single-Step Debugging

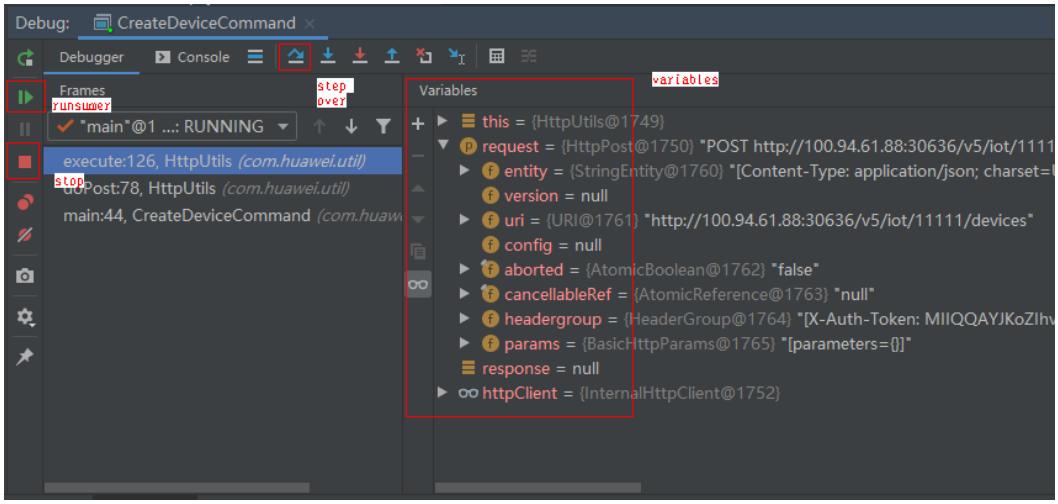
To intuitively view requests sent by applications and responses from the platform, use the breakpoint debugging method of IDEA.

Step 1 Set breakpoints in the code where HTTP or HTTPS messages are sent. For example, set three breakpoints for the **execute** method in the sample code **HttpsUtil.java**. (Set the breakpoints based on your actual code.)


```
private HttpResponse execute(HttpUriRequest request) {

    CloseableHttpResponse response = null;
    try {
        response = httpClient.execute(request);
    } catch (IOException e) {
        System.out.println(e);
    } finally {
        try {
            return new StreamClosedHttpResponse(response);
        } catch (IOException e) {
            System.out.println(e);
        }
```

Step 2 Right-click the class to debug, for example, **CreateDevice.java**, and choose **Debug** > **CreateDevice.main()**.

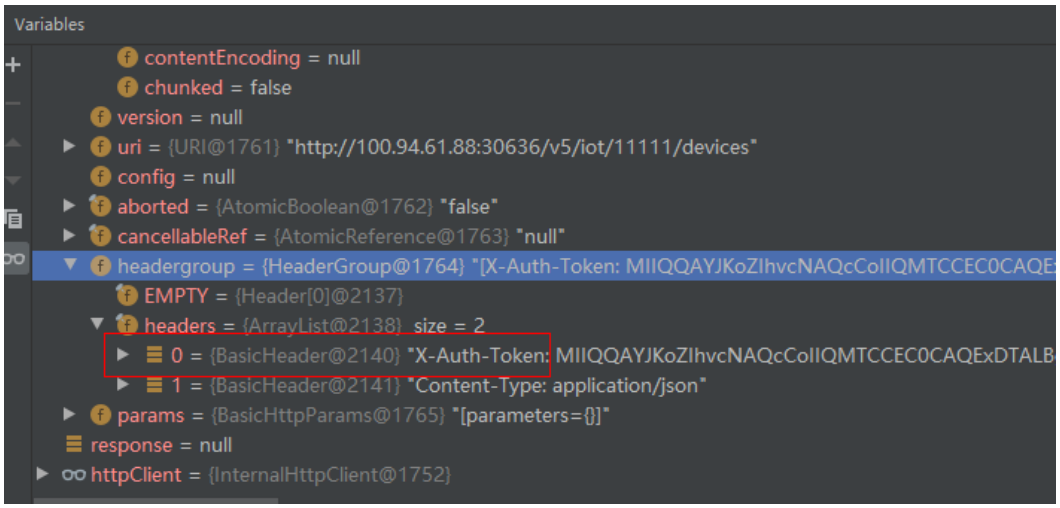
Step 3 After the program stops running at the breakpoint, click **Step Over** to perform single-step debugging. You can view the content of the variables in the **Variables** window, such as the **request** and **response**.



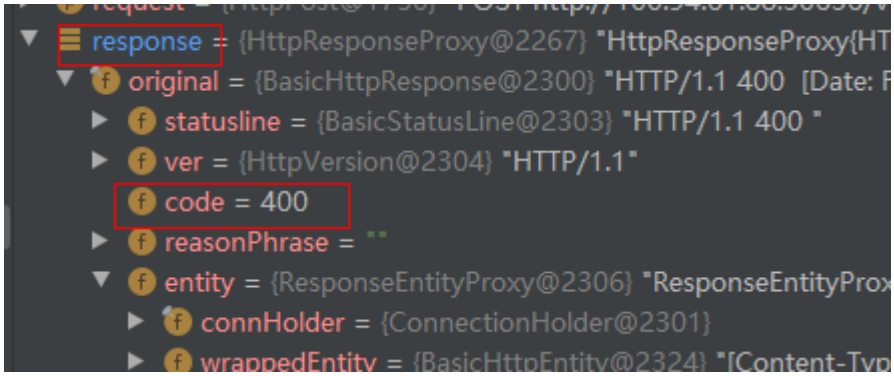
Step 4 Expand the **request** variable in the **Variables** window to view the content.

1. When the **request** variable is selected, the URL of the request sent by the application is displayed in the **uri** area, and the content of the request is displayed in the **entity** area.

2. The token is carried in **headerGroup**.



Step 5 Expand the **response** variable in the **Variables** window to view the content.



In the sample code, all classes other than **Authentication.java** call the Authentication API in the first step. Therefore, if you want to obtain a new token during single-step debugging on a class other than **Authentication.java**, view the variable content when the program reaches the breakpoint for the second time.

----End

5.4 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, read **API Reference on the Application Side** in advance. 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 to connect an application 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**

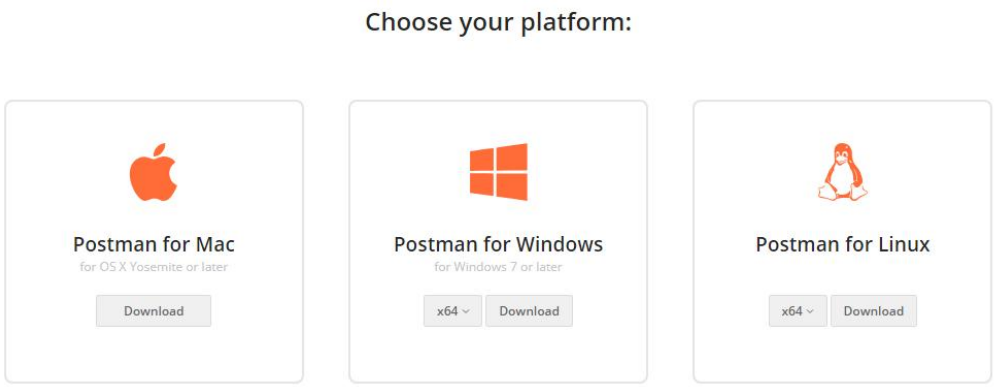
Prerequisites

- 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 **codec** on the IoTDA console.

Installing and Configuring Postman

Step 1 Install Postman.

1. Visit the **Postman website** to download and install Postman. (Postman 7.17.0 is used as an example.)



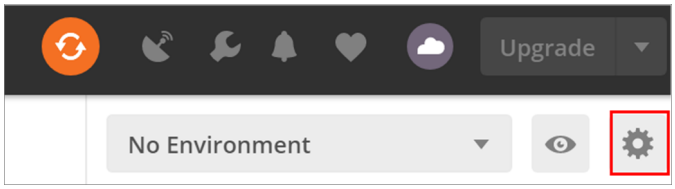
NOTE

- Postman requires the **.NET Framework 4.5** component. If you do not have this component, click **.NET Framework 4.5** to download and install it.
- To ensure successful API calls, you are advised to download Postman 7.17.0.

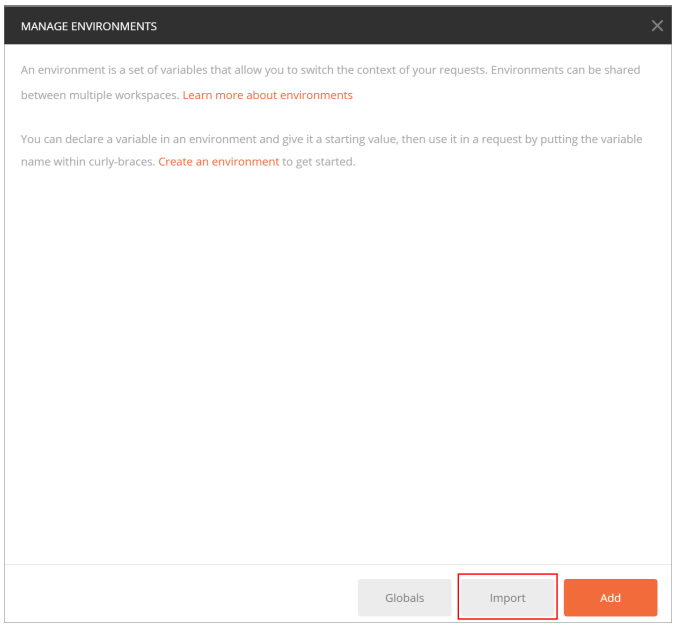
2. Enter the email address, username, and password to register Postman.

Step 2 Import the Postman environment variables.

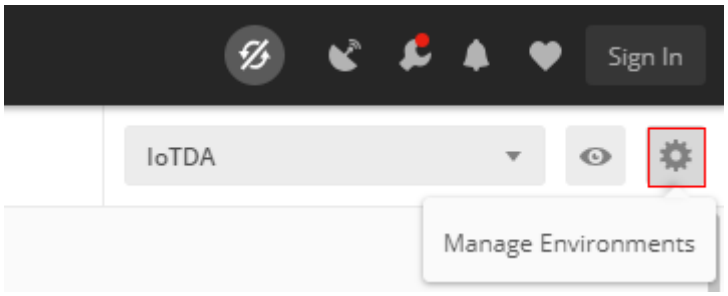
1. Click  in the upper right corner. The **MANAGE ENVIRONMENTS** window is displayed.

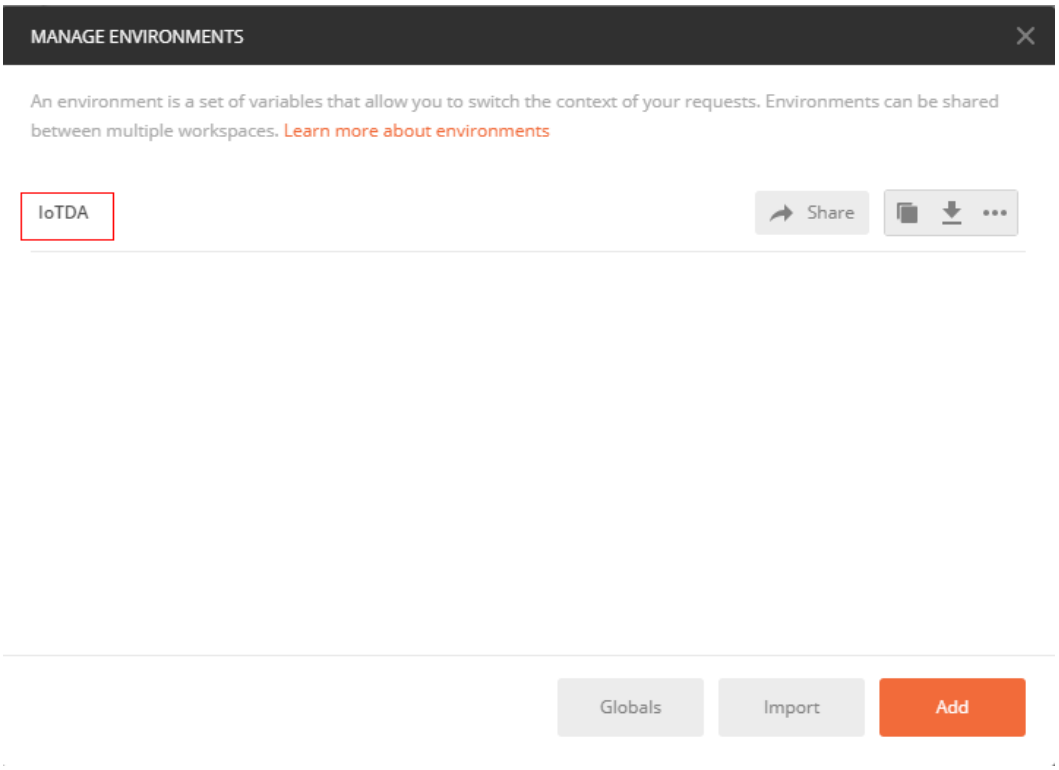


2. Click **Import** to import the **IoTDA.postman_environment.json** file (obtained after the **Collection** package is decompressed).

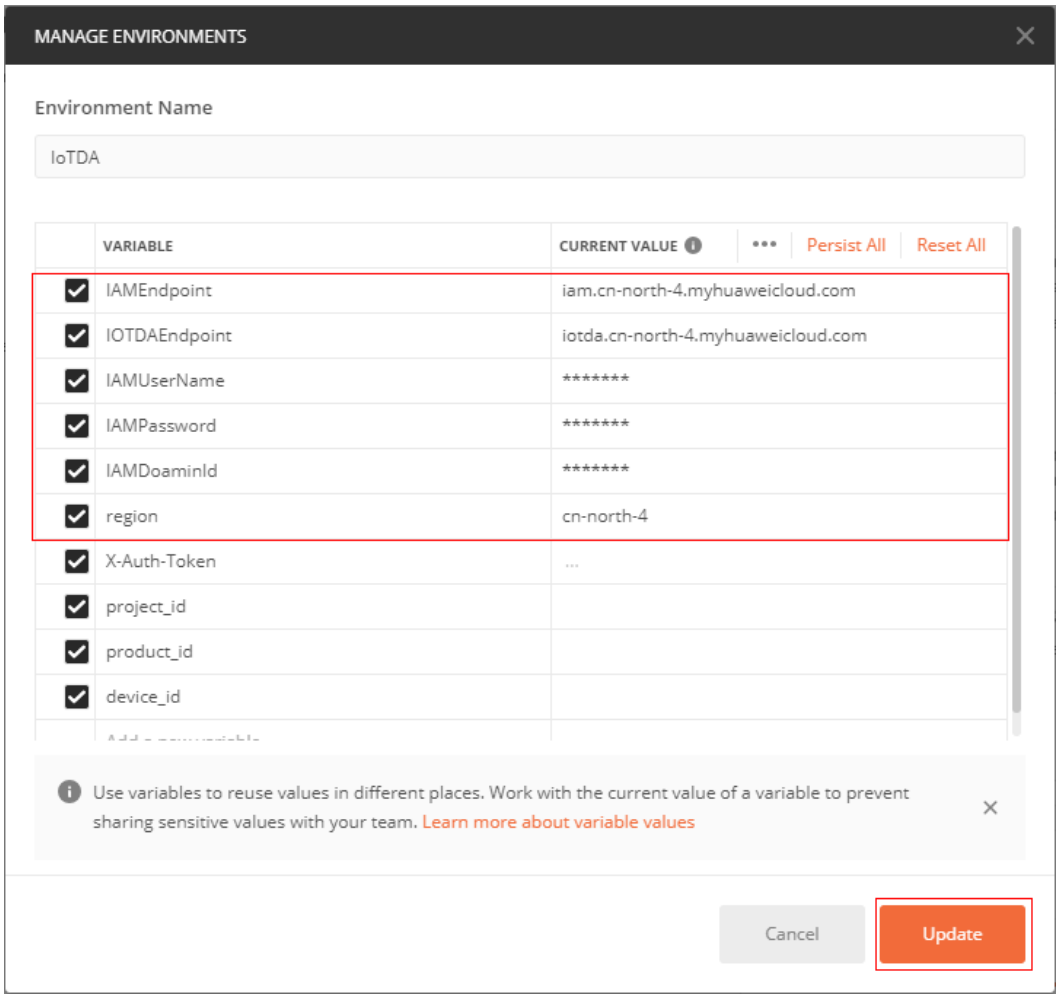


3. Click **Manage Environments** and select the imported IoTDA environment.





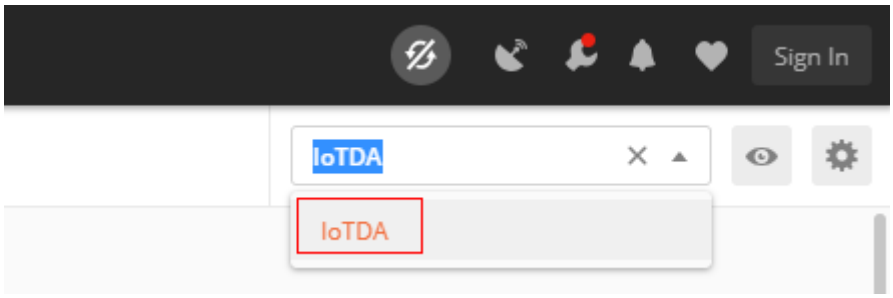
4. Change the values of **IAMEndpoint**, **IOTDAEndpoint**, **IAMUserName**, **IAMPassword**, **IAMDoaminId**, and **region**.



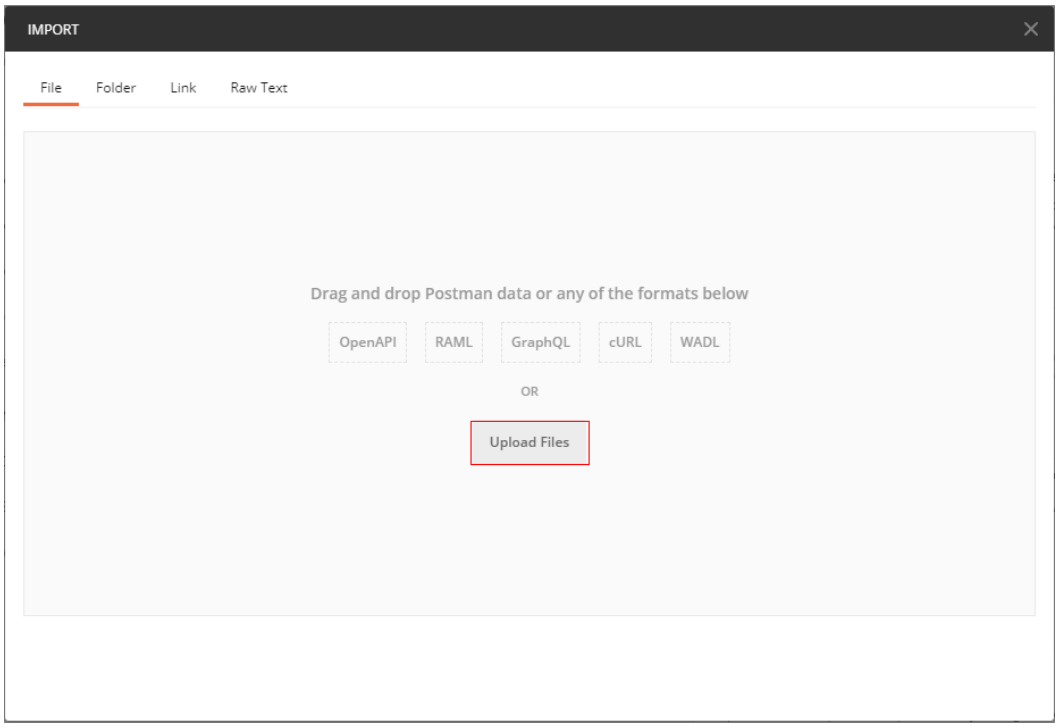
NOTE

- **IAMEndpoint**: Obtain the IAM endpoint from **IAM Regions and Endpoints**.
- **IOTDAEndpoint**: Obtain the IAM endpoint from **IoT Platform Endpoints**.
- If you have subscribed to IoTDA in CN North-Beijing4, change the IAM user name, login password, and account name by following the instructions provided in **My Credentials**.

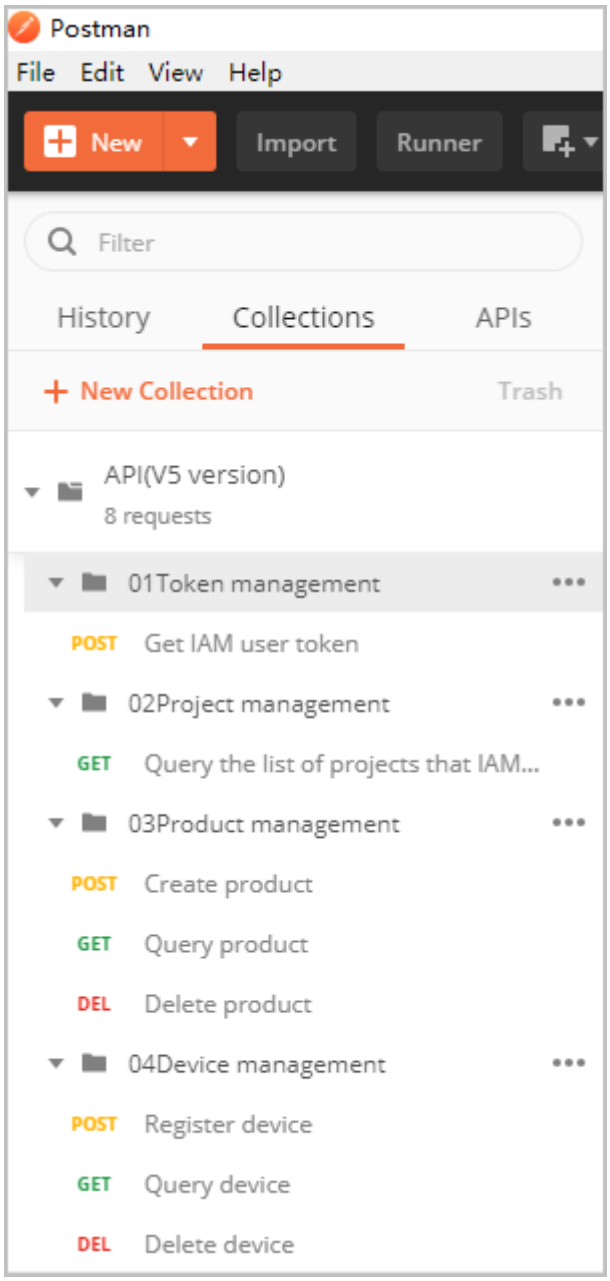
5. Return to the home page and set the environment variable to the imported IoTDA.



Step 3 Upload the **API call (V5 version).postman_collection.json** file.



After the file is uploaded, the dialog box shown in the following figure is displayed.



----End

Debugging the API Used to Obtain the Token for an IAM User

Before accessing platform APIs, an application must call the API **Obtaining the Token for an IAM User** for authentication. After the authentication is successful, HUAWEI CLOUD returns the authentication token **X-Subject-Token** to the application.

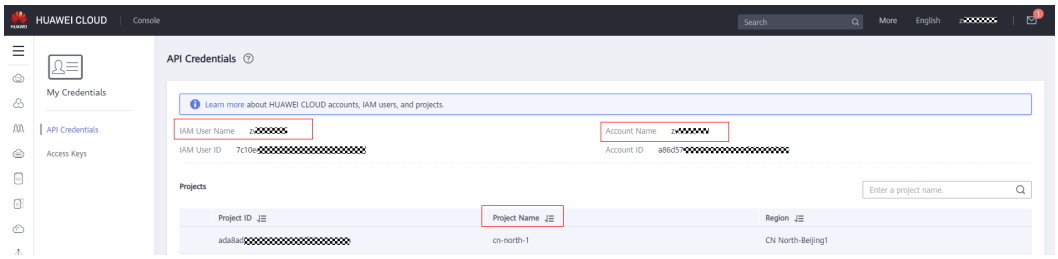
To call this API, the application constructs an HTTP request. An example request is as follows:

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

{
```



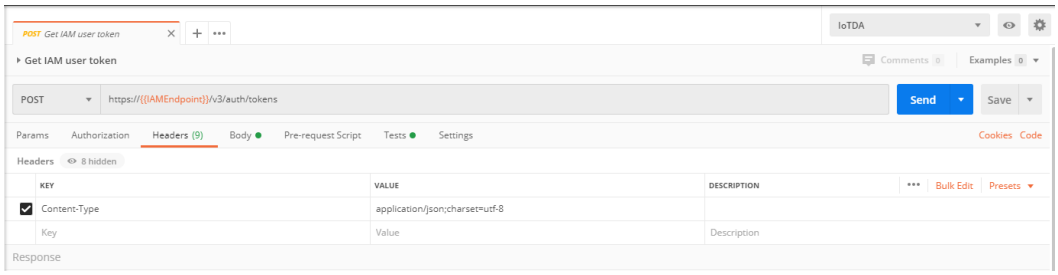
```
  "auth": {
    "identity": {
      "methods": [
        "password"
      ],
      "password": {
        "user": {
          "name": "username",
          "password": "********",
          "domain": {
            "name": "domainname"
          }
        }
      }
    },
    "scope": {
      "project": {
        "name": "xxxxxxxx"
      }
    }
  }
}
```

Note: **username** indicates the IAM user name, **password** indicates the password for logging in to HUAWEI CLOUD, **domainname** indicates the account name, and **projectname** indicates the project name. You can obtain them from the **My Credentials** page.

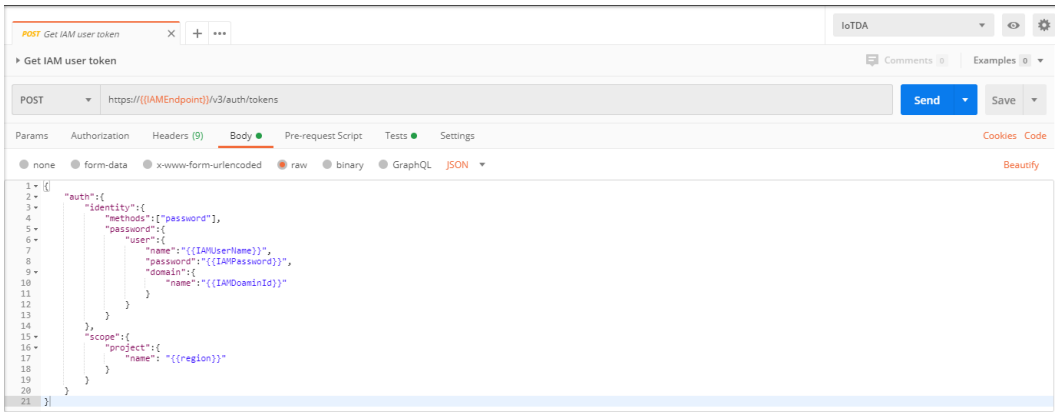


Debug the API by following the instructions provided in **Obtaining a User Token Through Password Authentication**.

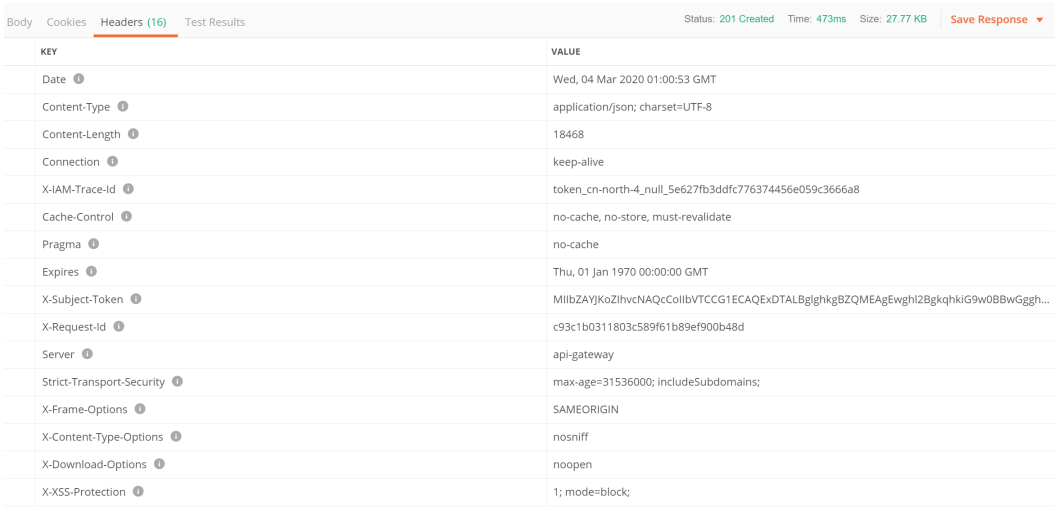
Step 1 Configure the HTTP method, URL, and headers of the API.



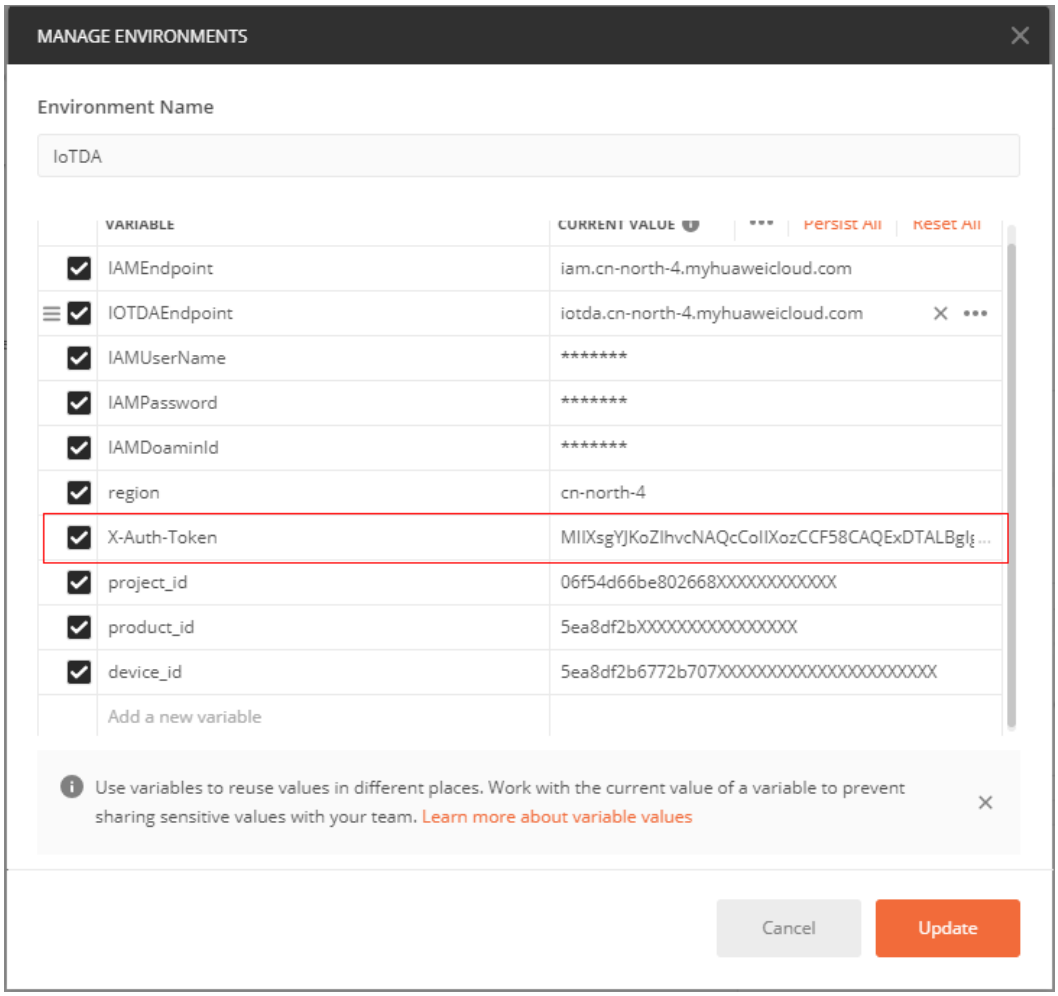
Step 2 Configure the body of the API.



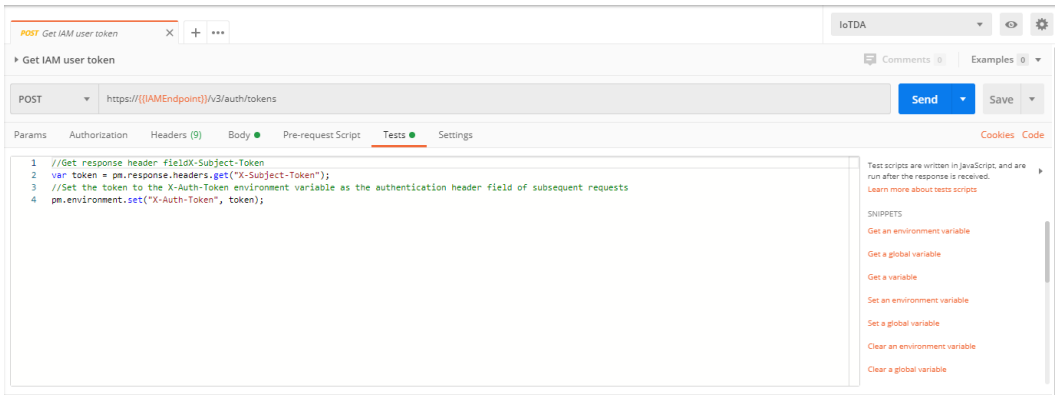
Step 3 Click **Send**. The returned code and response are displayed in the lower part of the page.



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

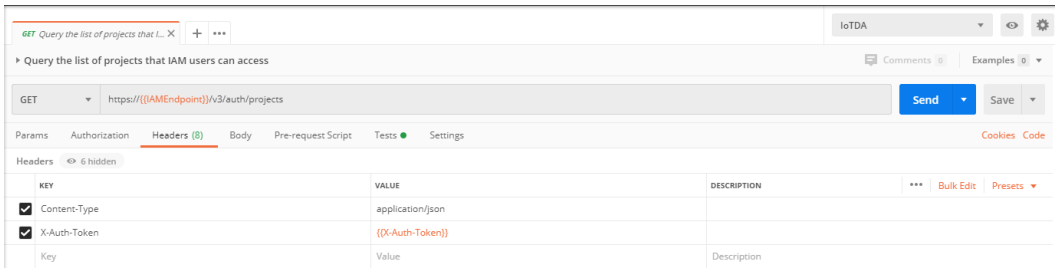
Before accessing platform APIs, the application must call the API **Listing Projects Accessible to an IAM User** to obtain the project ID of the user.

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

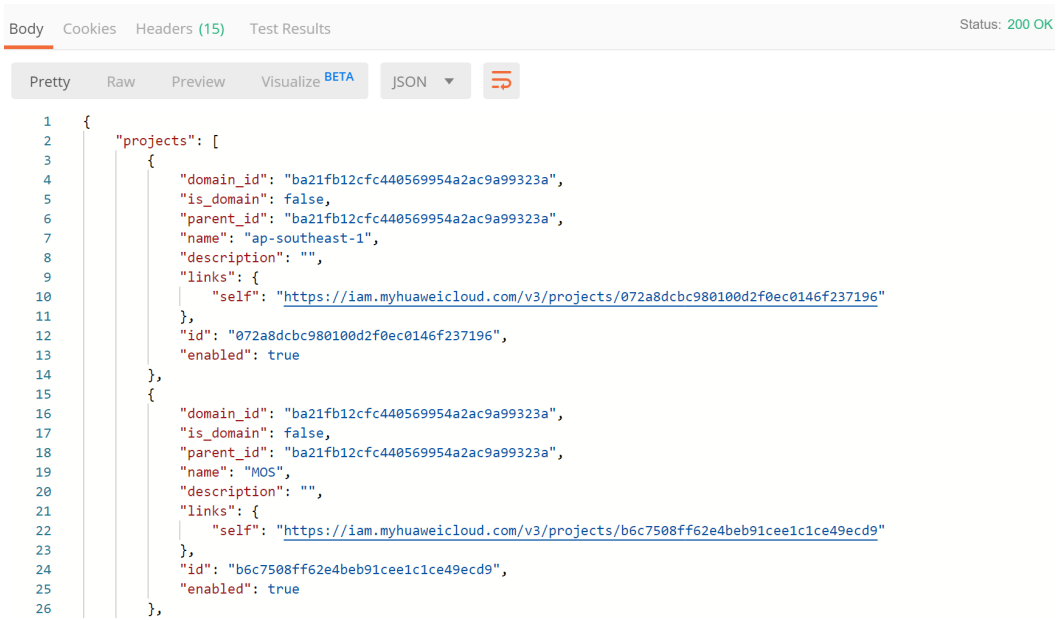

```
GET https://iam.cn-north-4.myhuaweicloud.com/v3/auth/projects
Content-Type: application/json
X-Auth-Token: ********
```

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.

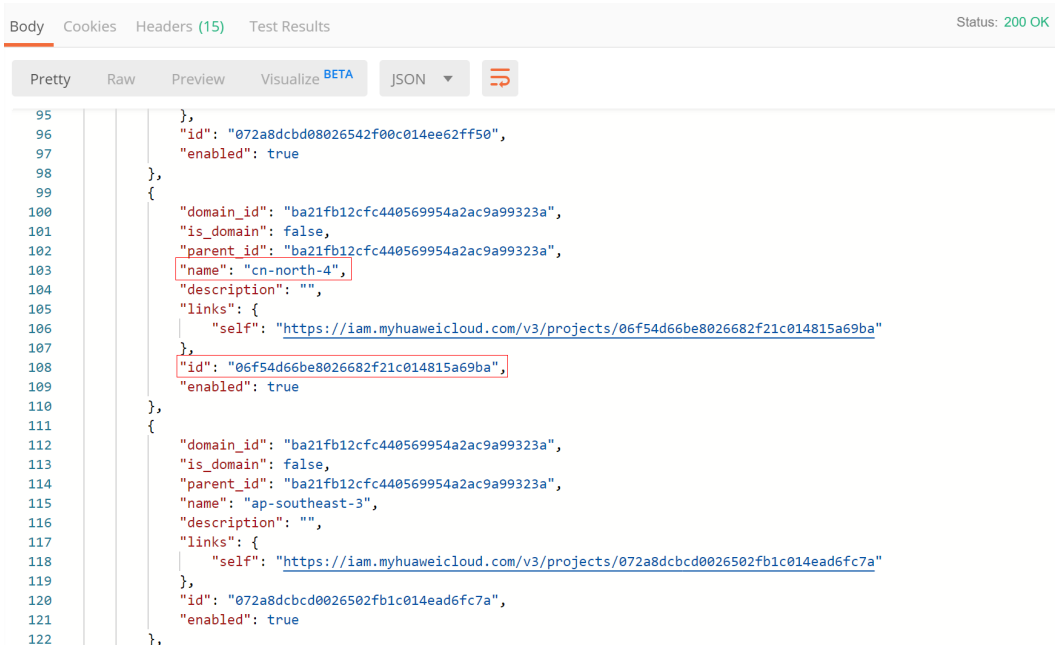


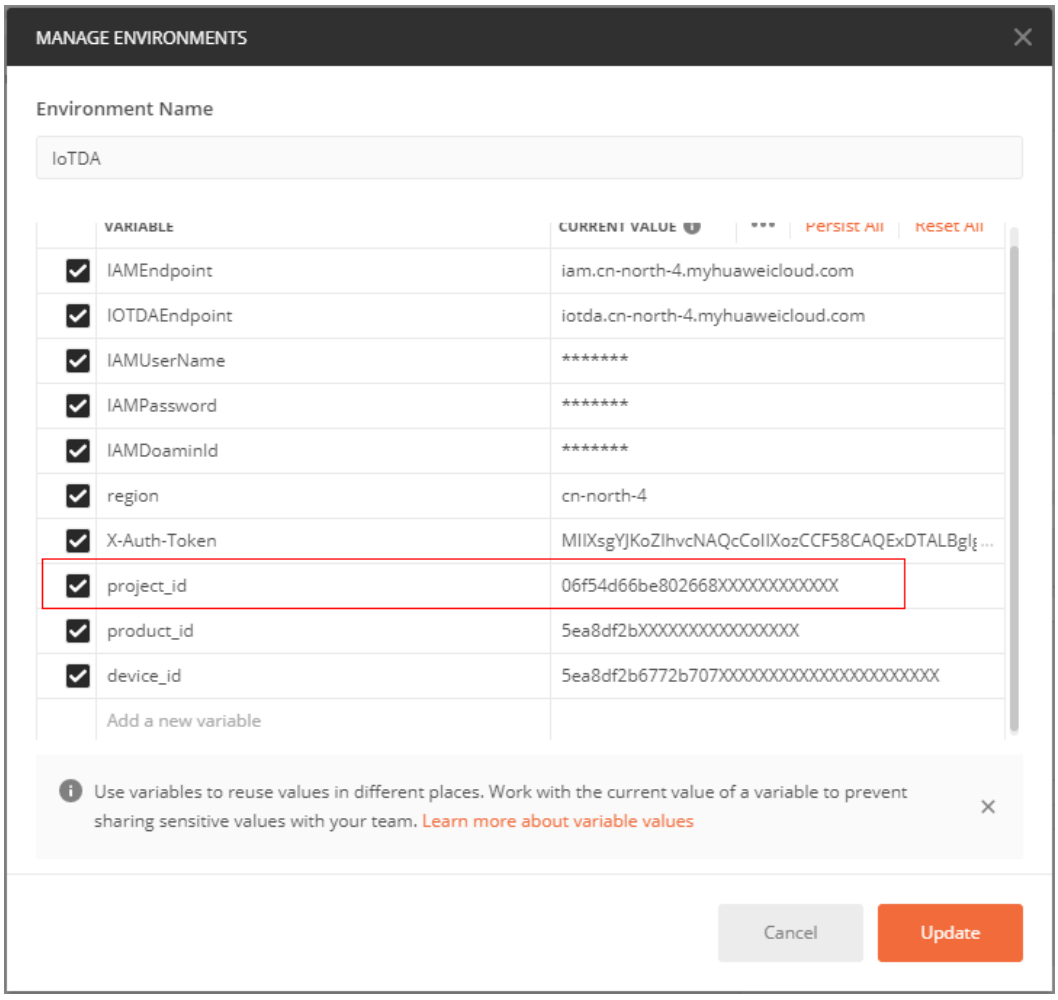
Step 2 Click **Send**. The returned code and response are displayed in the lower part of the page.



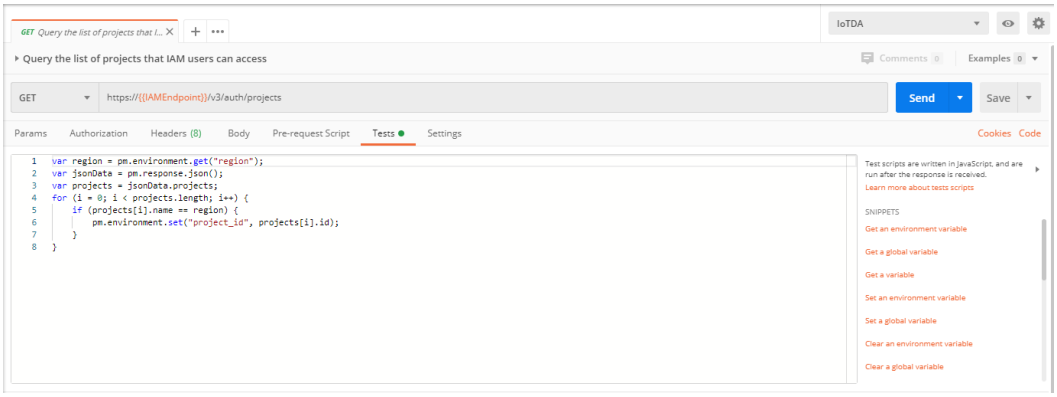

```
{
    "projects": [
        {
            "domain_id": "ba21fb12cfc440569954a2ac9a99323a",
            "is_domain": false,
            "parent_id": "ba21fb12cfc440569954a2ac9a99323a",
            "name": "ap-southeast-1",
            "description": "",
            "links": {
                "self": "https://iam.myhuaweicloud.com/v3/projects/072a8dcbc980100d2f0ec0146f237196"
            },
            "id": "072a8dcbc980100d2f0ec0146f237196",
            "enabled": true
        },
        {
            "domain_id": "ba21fb12cfc440569954a2ac9a99323a",
            "is_domain": false,
            "parent_id": "ba21fb12cfc440569954a2ac9a99323a",
            "name": "MOS",
            "description": "",
            "links": {
                "self": "https://iam.myhuaweicloud.com/v3/projects/b6c7508ff62e4beb91cee1c1ce49ecd9"
            },
            "id": "b6c7508ff62e4beb91cee1c1ce49ecd9",
            "enabled": true
        },
```

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 Used to Create 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,
```

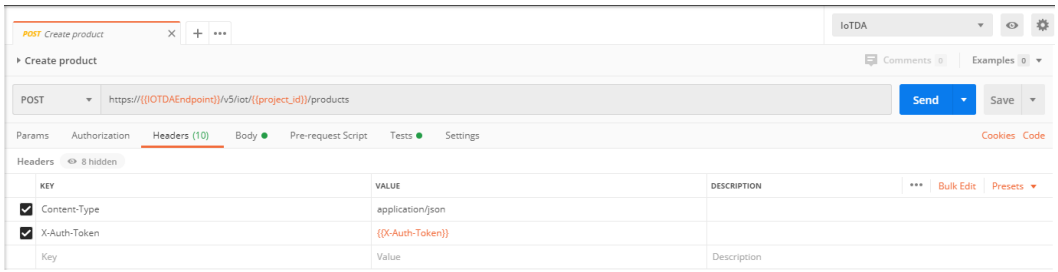


```
        "enum_list" : [ "string" ],
        "required" : false,
        "max_length" : 100
      } ]
    } ],
    "paras" : [ {
      "unit" : "km/h",
      "min" : "1",
      "max" : "100",
      "para_name" : "force",
      "data_type" : "string",
      "description" : "force",
      "step" : 0.1,
      "enum_list" : [ "string" ],
      "required" : false,
      "max_length" : 100
    } ]
  } ],
  "option" : "Mandatory"
 } ],
 "app_id" : "jeQDJQZltU8iKgFFoW060F5SGZka"
}
```

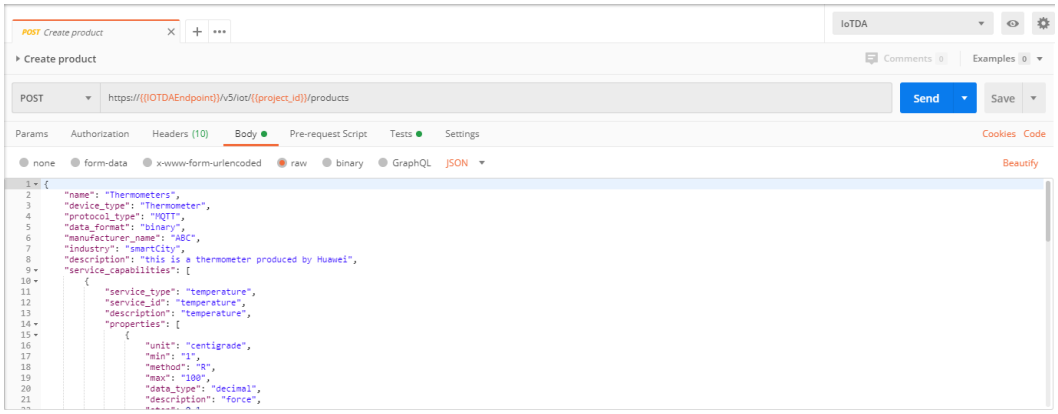
Debug the API by following the instructions provided in **Creating a Product**.

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

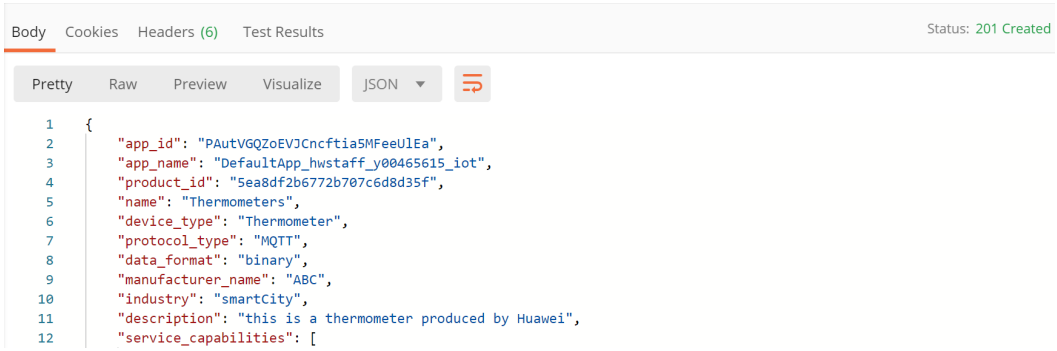
Step 1 Configure the HTTP method, URL, and headers of the API.



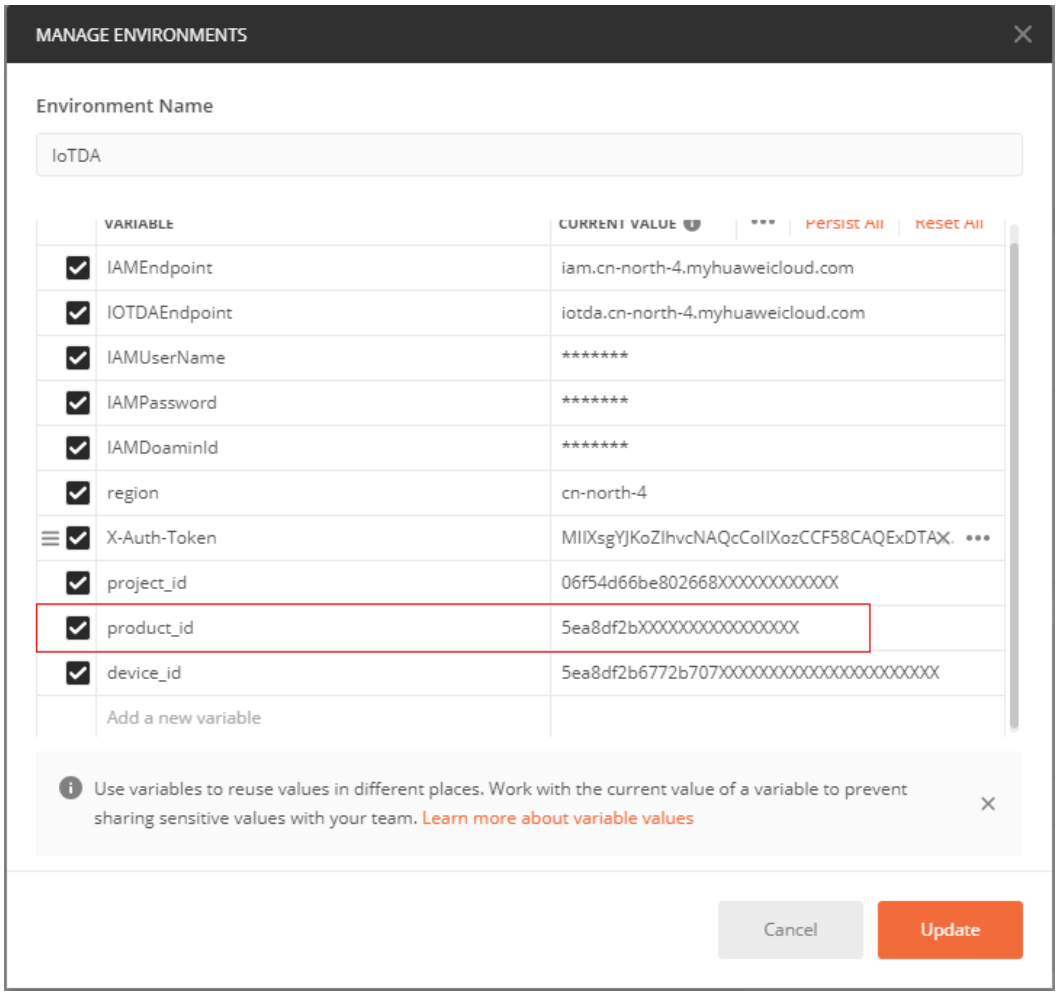
Step 2 Configure the body of the API.



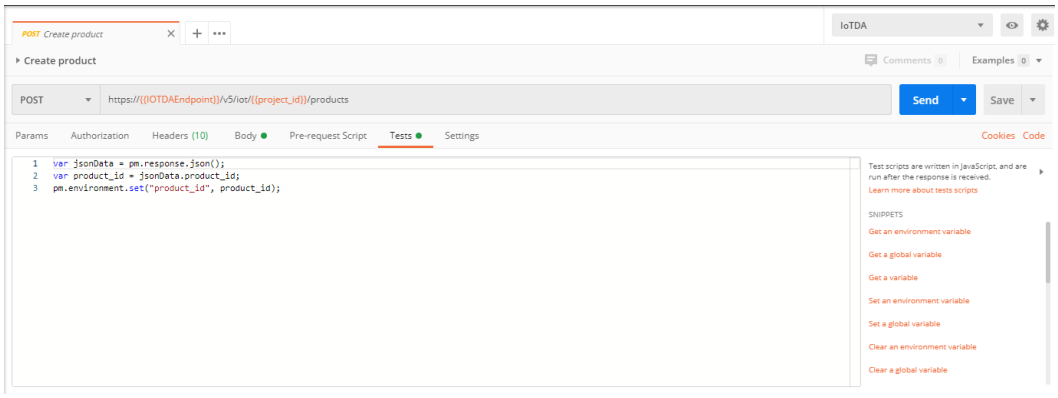
Step 3 Click **Send**. The returned code and response are displayed in the lower part of the page.



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.

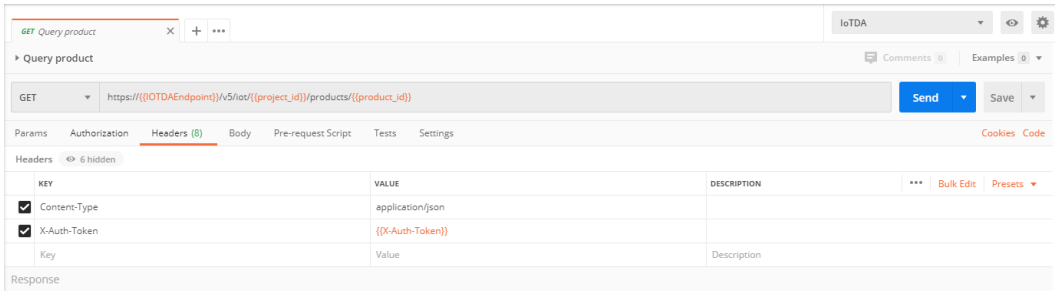
To call this API, the application constructs an HTTP request. An example request is as follows:

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

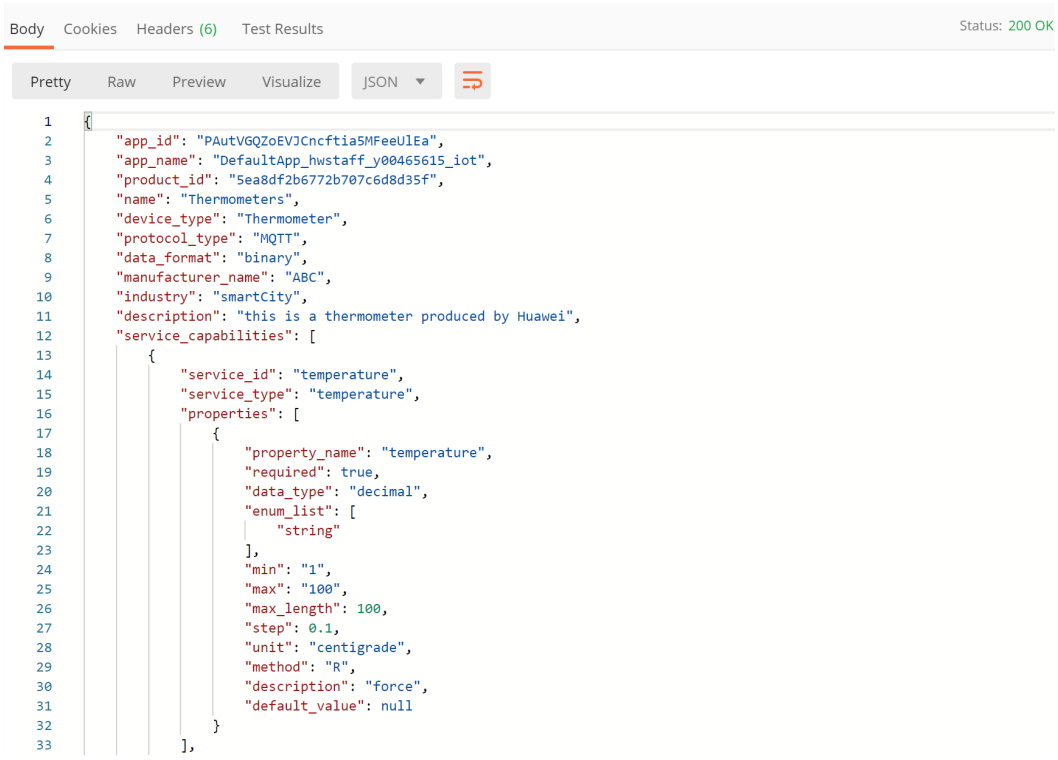
Debug the API by following the instructions provided in **Querying a Product**.

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.

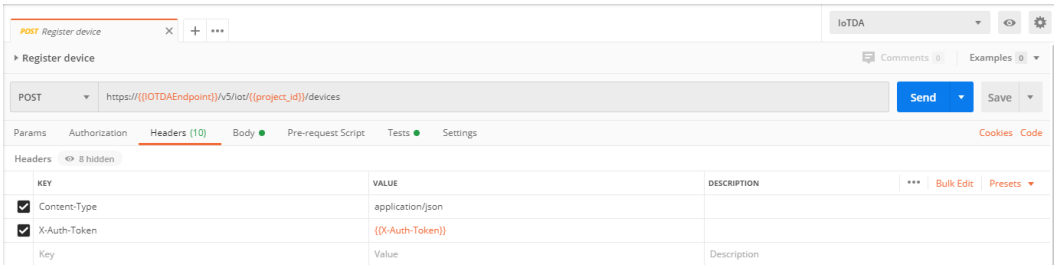
To call this API, the application constructs an HTTP request. An example request is as follows:

```
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"
}
```

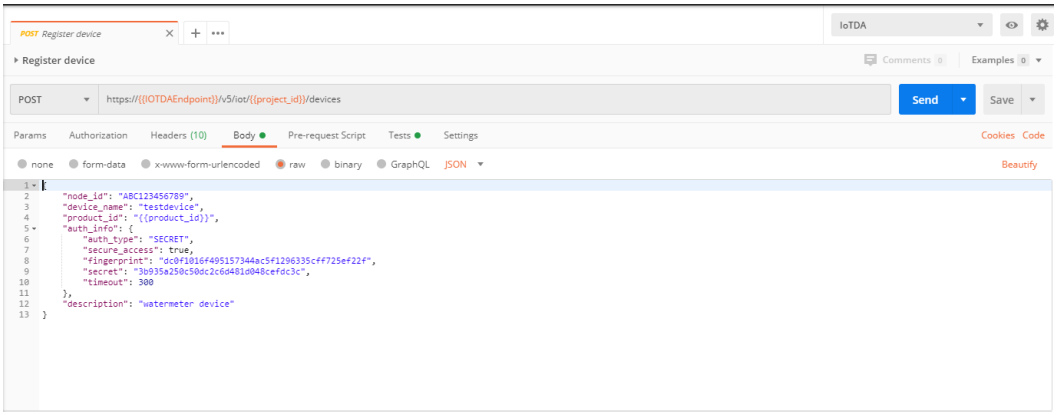
Debug the API by following the instructions provided in **Creating a Device**.

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

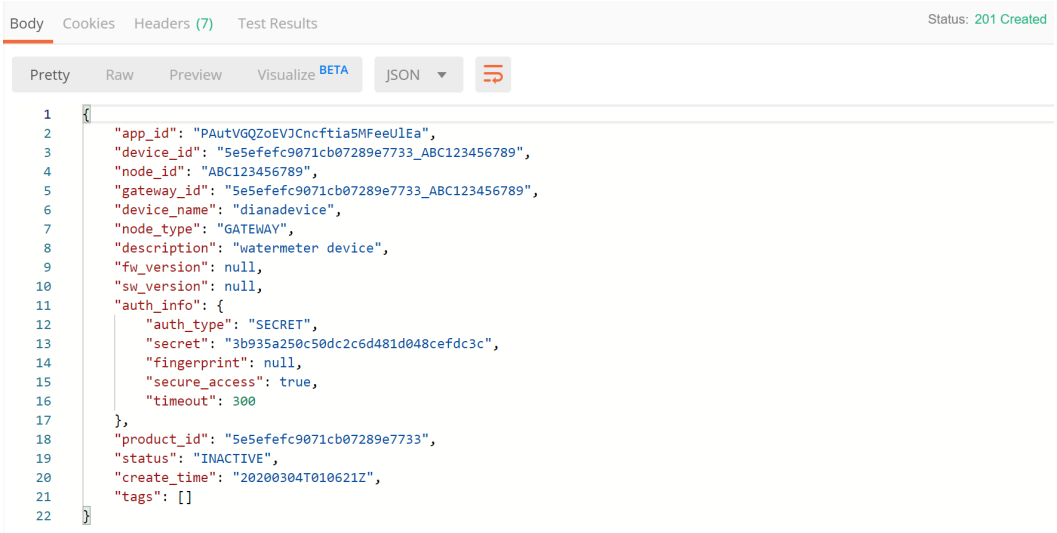
Step 1 Configure the HTTP method, URL, and headers of the API.



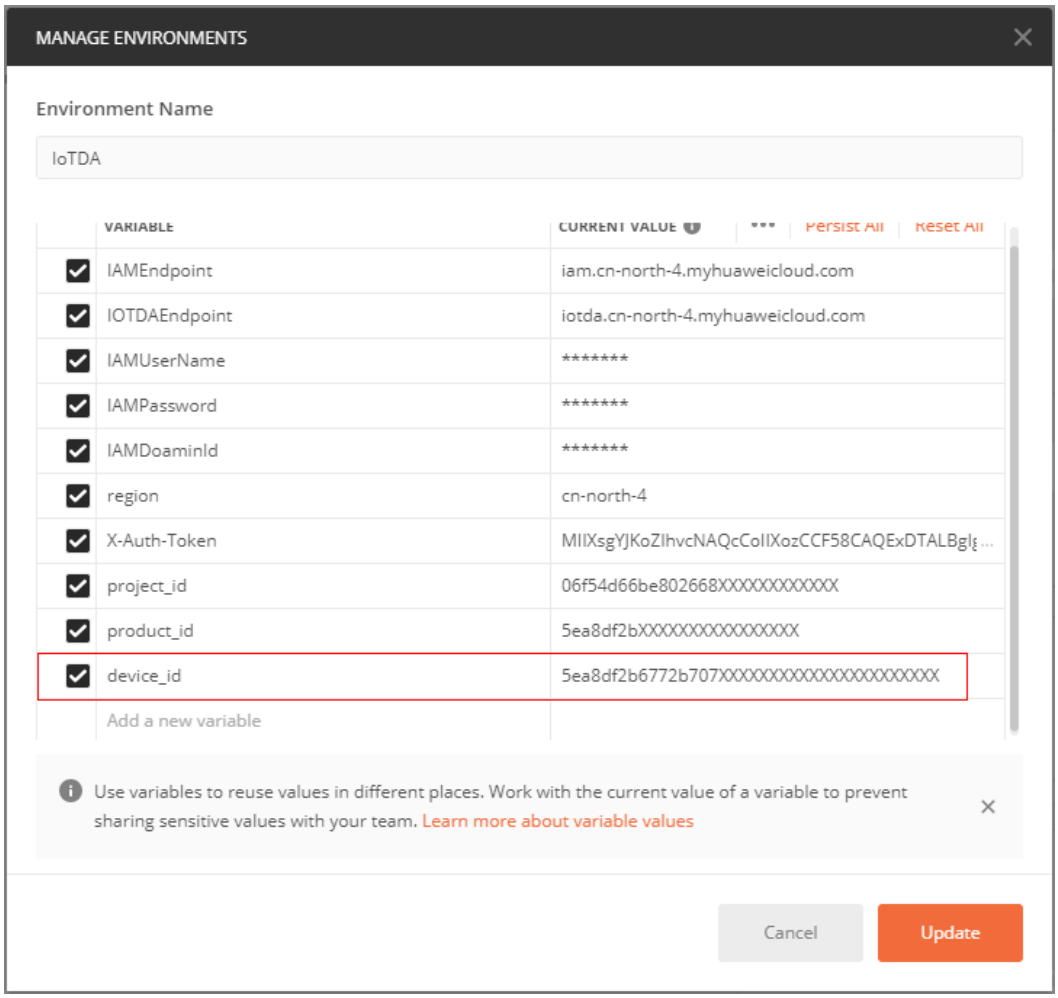
Step 2 Configure the body of the API.



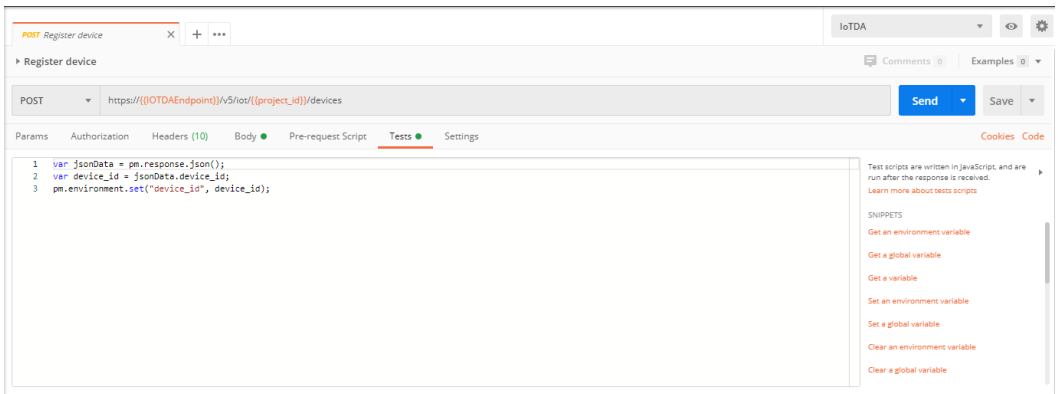
Step 3 Click **Send**. The returned code and response are displayed in the lower part of the page.



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.

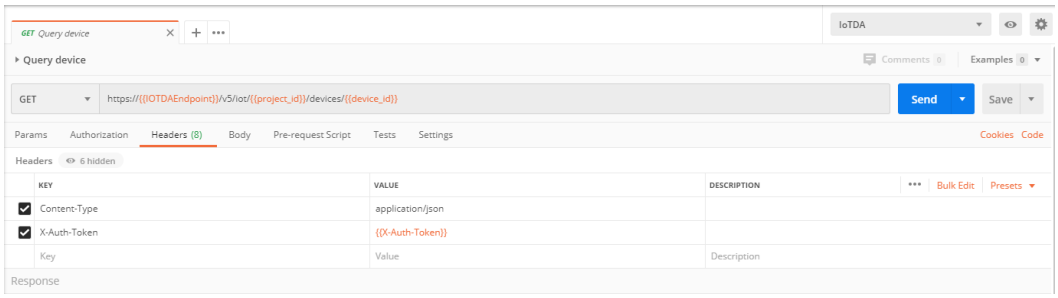
To call this API, the application constructs an HTTP request. An example request is as follows:


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

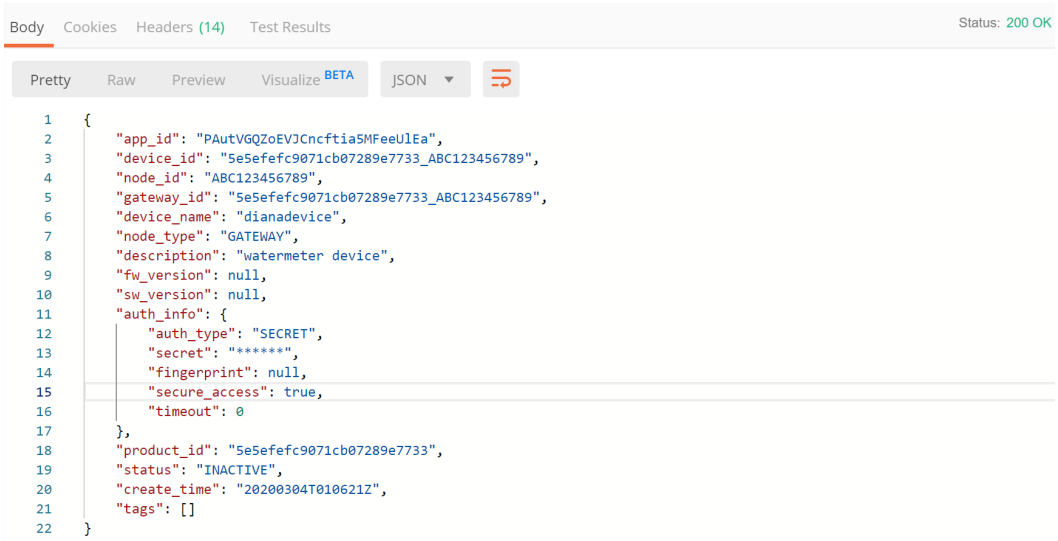
Debug the API by following the instructions provided in **Querying a Device**.

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

