IoT Device Management

Development Guide

Issue	02
Date	2019-08-28

HUAWEI

HUAWEI TECHNOLOGIES CO., LTD.

Contents

1 Product Development

1.1 Obtaining Development Resources

Application Development Resources

The IoT platform provides a wealth of RESTful APIs and SDKs to ease application development. Application development is the process in which an application calls APIs of the IoT platform to implement service scenarios such as secure access, device management, data collection, and command delivery. Download the corresponding resource files as required.

Resource Package	Description	Download Link
Application Development Java API Demo	The IoT platform provides the RESTful API for application developers to quickly experience open API capabilities, service functions, and service processes. For details, see **Northbound API Reference** and **Application Development Guide**.	**Application Development Java API Demo**
Application Development Java SDK	The Java SDK provides Java methods to call RESTful APIs to communicate with the IoT platform. The Java SDK Demo provides the code sample for calling the SDK APIs. For details, see **Northbound Java SDK API Reference** and **Java SDK Usage Guide**.	● **JAVA SDK** ● **JAVA SDK Demo**

Device Development Resources

The IoT platform allows device access using MQTT or LWM2M/CoAP. Devices can connect to the IoT platform by calling device APIs or integrating with SDKs.

Resource Package	Description	Download Link
LiteOS SDK	Devices can connect to the IoT platform through the integrated LiteOS SDK. The LiteOS Demo provides the code sample for calling the SDK APIs. For details, see **LiteOS SDK Integration Development Guide**.	**LiteOS SDK**
Profile Templates	Profile templates of typical scenarios are provided. Developers can customize their profile files based on the templates. For details, see **Offline Profile Definition**.	**Profile Example**

Resource Package	Description	Download Link
Codec Example	Demo codec projects are provided for developers to perform secondary development. For details, see **Offline Codec Development**.	**Codec Example**
Codec Test Tool	This tool is used to check whether the codec developed offline is normal.	**Codec Test Tool**
NB-IoT Device Simulator	This tool is used to simulate the access of NB-IoT devices to the IoT platform using CoAP for data reporting and command delivery.	**NB-IoT Device Simulator**

Certificates

In some scenarios where a device and NA connect to the IoT platform, the corresponding certificate must be loaded to the device and NA. Click **here** to obtain the certificate files.

NOTE

This certificate package is used only for interconnection with the IoT platform deployed on HUAWEI CLOUD.

For details about the directory structure of the certificate package and the usage of each certificate, see **Table 1-1**.

Table 1-1 Certificate information

Certificate Package Name	Level-1 Directory	Level-2 Directory	Level-3 Directory	Description
certificate	Northbound API	code	Java	The certificates in this directory are used when the NA calls IoT platform APIs using HTTPS. Select the certificate in the corresponding directory based on the programming language of the NA, and load the certificate to the NA.
			PHP	
			Python	
		postman	-	The certificate in this directory is used when Postman tests the IoT platform APIs using HTTPS.
	Agent Lite	Android	-	The certificates in this directory are used when the device or gateway connects to the

Certificate Package Name	Level-1 Directory	Level-2 Directory	Level-3 Directory	Description
		C-Linux	-	IoT platform through the integrated AgentLite SDK. Select the certificate in the corresponding directory based on the programming language of the device or gateway, and load the certificate to the device or gateway.
		Java	-	

1.2 Creating a Project and Product

Concept

- Project: an independent space where you can develop IoT products and applications.
- Product: a collection of devices with the same capabilities or features. In addition to physical devices, a product includes product information, product models (profile files), codecs, and test reports generated during IoT capability building.

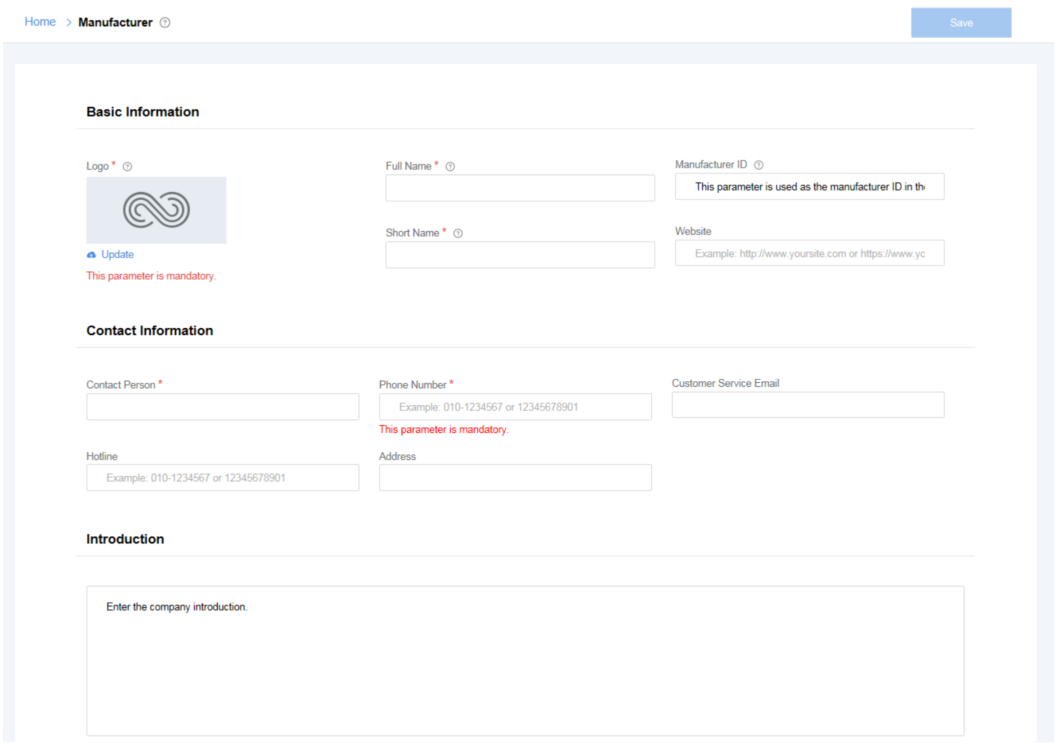
Procedure

Step 1 Access the home page of IoT Device Management, and click **Developer Center**.

NOTE

Currently, the Developer Center is available only in Hong Kong. You need to complete product development on the Developer Center in the Hong Kong region and then select your region to connect devices and applications.

Step 2 (Optional) If you are using the Developer Center for the first time, click **Manufacturer** in the upper right corner, complete the manufacturer information, and click **Save**.


Home > Manufacturer

Save

Basic Information

Logo *

Update

This parameter is mandatory.

Full Name *

Short Name *

Manufacturer ID

This parameter is used as the manufacturer ID in th

Website

Example: http://www.yoursite.com or https://www.yc

Contact Information

Contact Person *

Phone Number *

Example: 010-1234567 or 12345678901

This parameter is mandatory.

Customer Service Email

Hotline

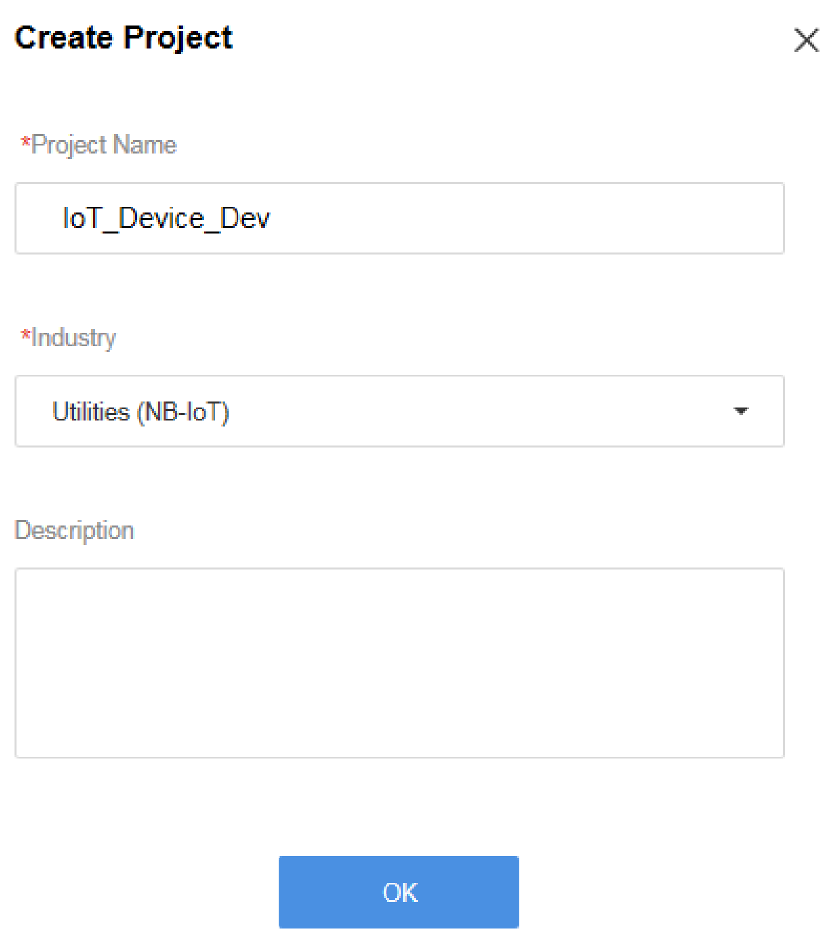
Example: 010-1234567 or 12345678901

Address

Introduction

Enter the company introduction.

Step 3 On the home page of the Developer Center, click **Create Project**. In the dialog box displayed, enter a project name, select the industry to which the project belongs, and click **OK**.



IoT_Device_Dev

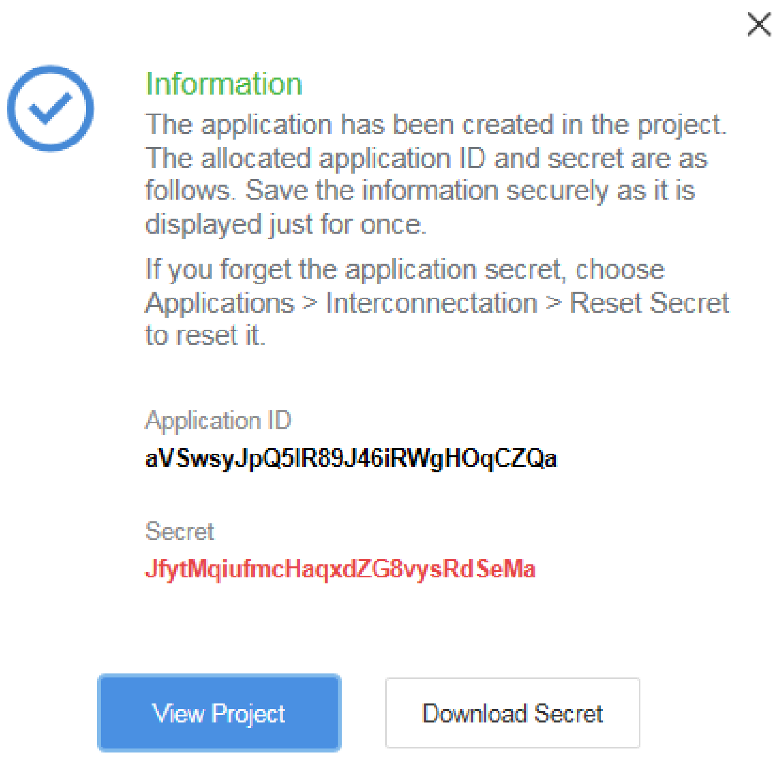
*Industry: Utilities (NB-IoT)

Description

Step 4 When a dialog box indicating that the project is created is displayed, click **Download Secret** to download the application ID and secret to your local PC, and click **View Project** to open the project.

NOTE

The application ID and secret are required when a network application (NA) accesses the IoT platform. Keep them securely. If you forget the secret, reset it by clicking **Reset Secret** under **Applications** > **Interconnection** > **Application Security**.


Information

The application has been created in the project. The allocated application ID and secret are as follows. Save the information securely as it is displayed just for once.

If you forget the application secret, choose Applications > Interconnectation > Reset Secret to reset it.

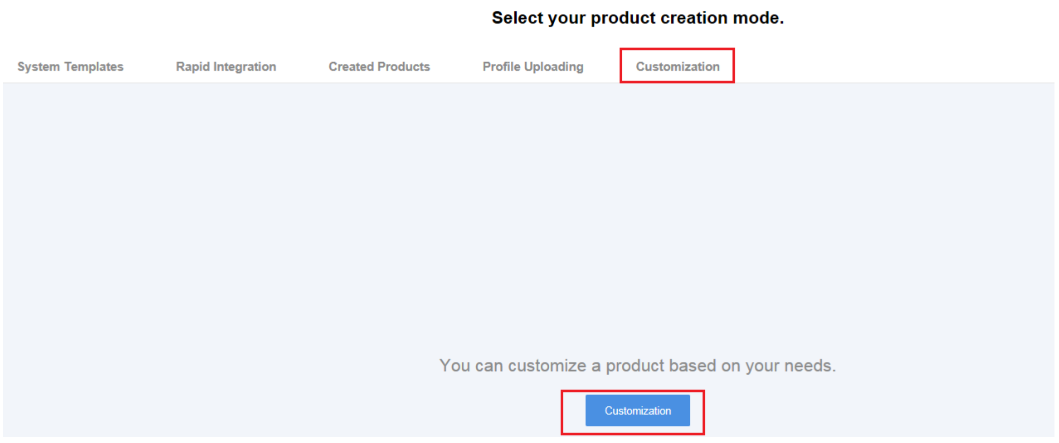
Application ID
aVSwsyJpQ5IR89J46iRWgHOqCZQa

Secret
JfytMqiufmcHaqxdZG8vysRdSeMa

Step 5 On the home page of the project, choose **Products** > **Product Development**, and click **Create Product**.

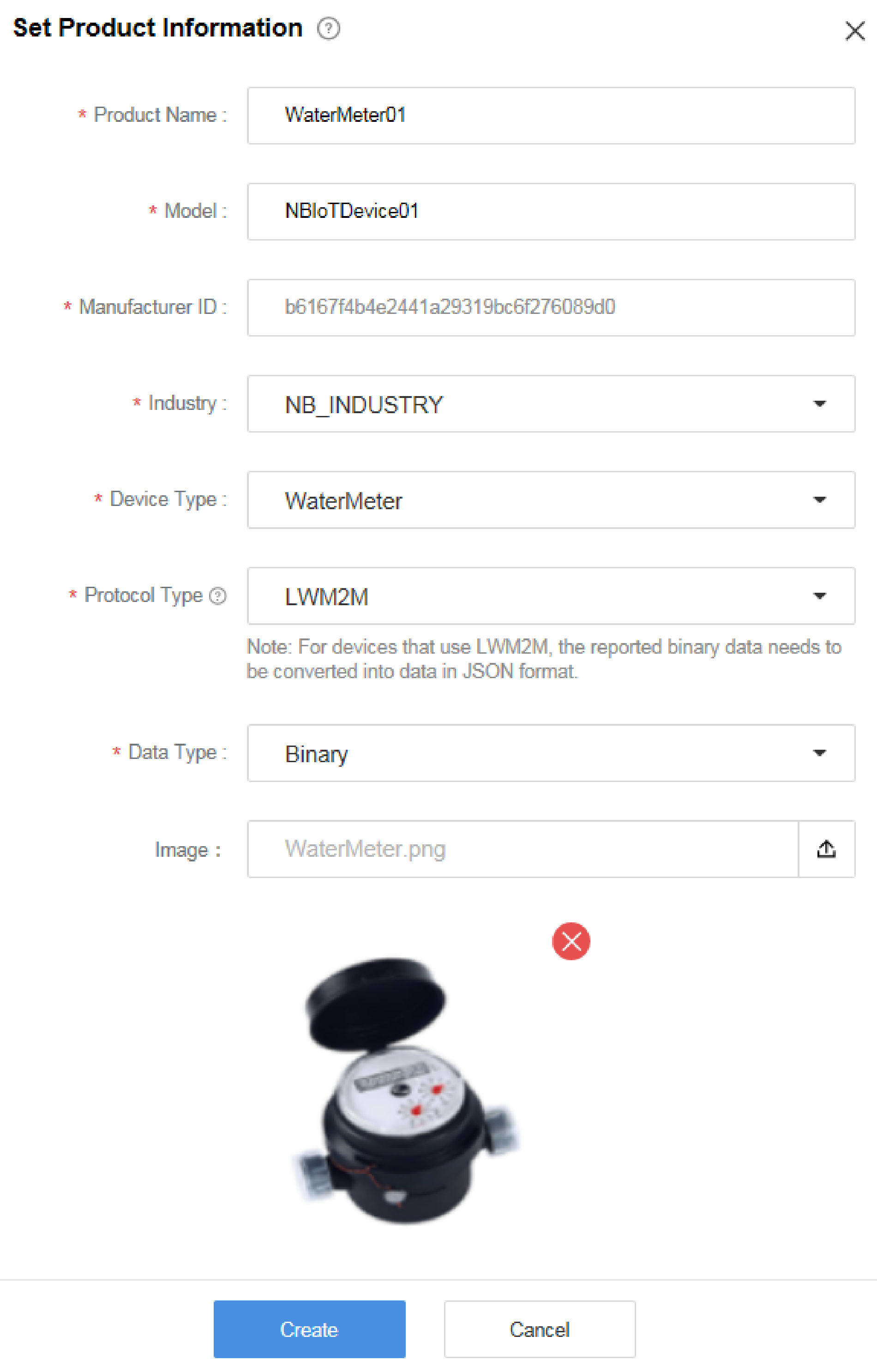
Step 6 You can create a product based on a preset template or customize a product. The following uses customization as an example.

Step 7 Click the **Customization** tab, and click **Customization**.


Select your product creation mode.

You can customize a product based on your needs.

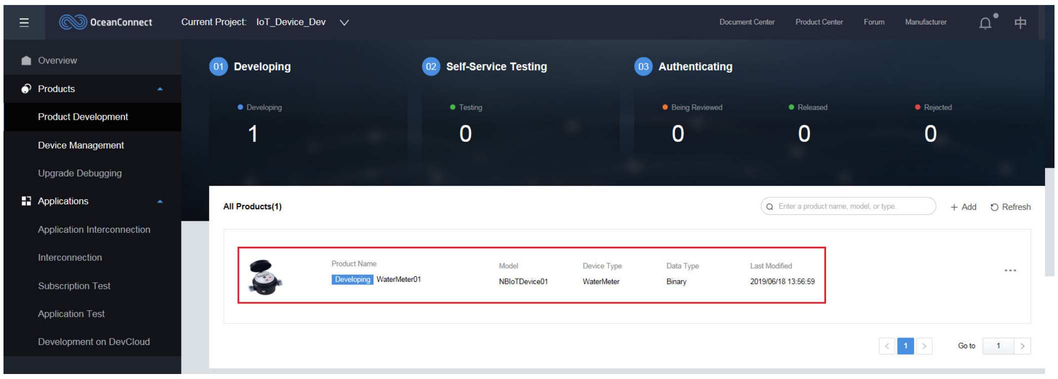
Step 8 In the **Set Product Information** dialog box, enter the information such as **Product Name**, **Model**, and **Industry**, and click **Create**.


Set Product Information

* Product Name :	WaterMeter01
* Model :	NBIoTDevice01
* Manufacturer ID :	b6167f4b4e2441a29319bc6f276089d0
* Industry :	NB_INDUSTRY
* Device Type :	WaterMeter
* Protocol Type	LWM2M
* Data Type :	Binary
Image :	WaterMeter.png

Note: For devices that use LWM2M, the reported binary data needs to be converted into data in JSON format.

Step 9 On the **Product Development** page, click the product to enter its development space.



Product Name	Model	Device Type	Data Type	Last Modified
Developing WaterMeter01	NBIoTDevice01	WaterMeter	Binary	2019/06/18 13:56:59

----End

1.3 Developing a Product Model

1.3.1 Development Guide

Overview

A product model (or profile file) describes the capabilities and features of a device. You can construct an abstract model of a device type by defining a profile file on the IoT platform, allowing it to understand the services, properties, and commands supported by the device.

- **Device Capability**

  For a water meter, the device capabilities include the type, manufacturer, model, protocol, and services to be provided.

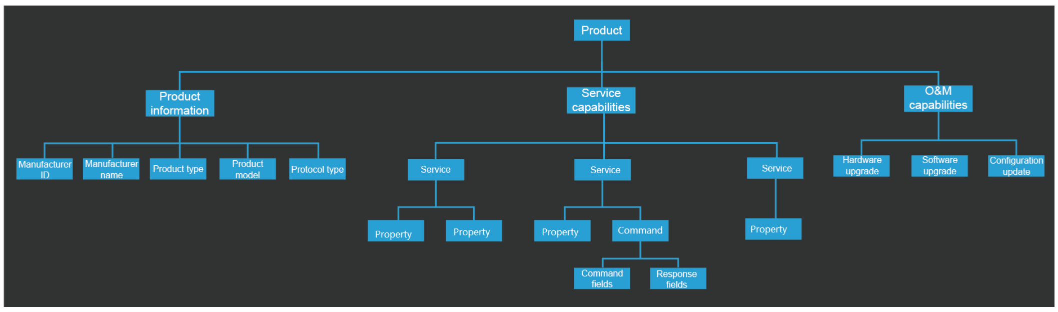
  For example, for a water meter, the manufacturer is **HZYB**, the manufacturer ID is **TestUtf8ManuId**, the model is **NBIoTDevice**, and the protocol is **CoAP**.

  The water meter provides the following services: WaterMeterBasic, WaterMeterAlarm, Battery, DeliverySchedule, and Connectivity. The Battery service is optional and the other services are mandatory.

- **Service**

  Service defines service capabilities of a device. Each service contains properties, commands, and parameters.

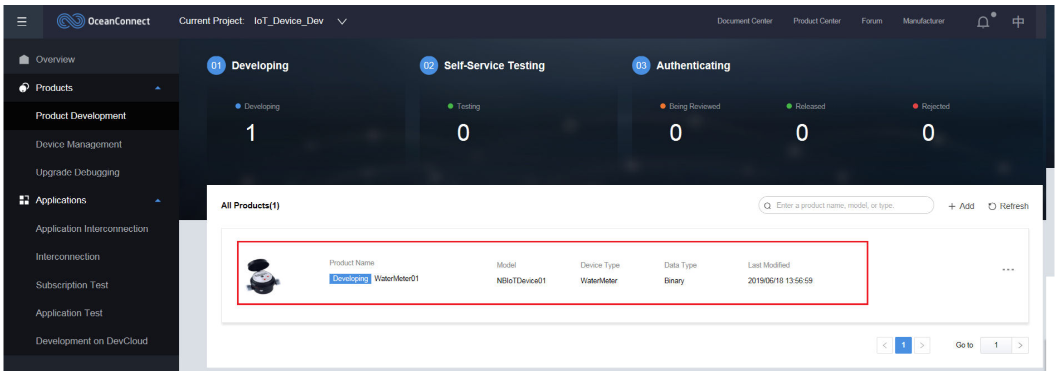
  For example, the preceding five services of the water meter contain corresponding properties or commands.



Procedure

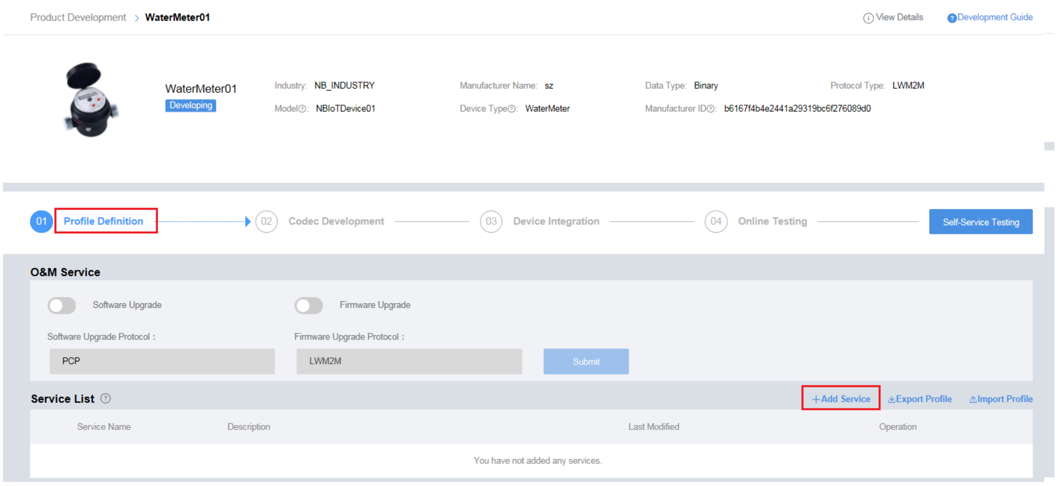
If you have used a default template when **creating a project and product**, the corresponding profile file template is selected automatically. You can directly use or modify the template. If a customized product is created, you must define your profile file.

Step 1 On the **Product Development** page, click a product to enter its development space.



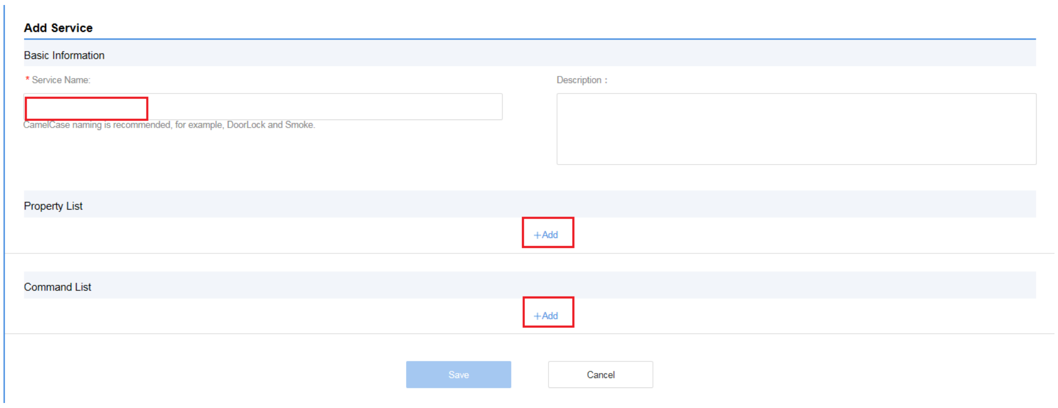
Product Name: Developing WaterMeter01; Model: NBIoTDevice01; Device Type: WaterMeter; Data Type: Binary; Last Modified: 2019/06/18 13:56:59

Step 2 In the development space, click **Profile Definition** and click **Add Service**.

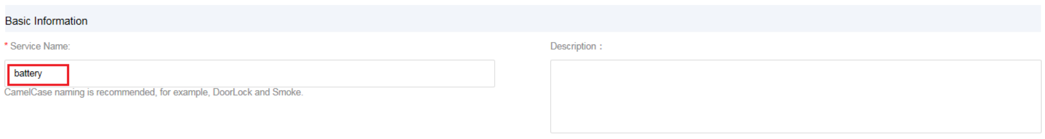


WaterMeter01 Developing
Industry: NB_INDUSTRY
Model: NBIoTDevice01
Manufacturer Name: sz
Device Type: WaterMeter
Data Type: Binary
Manufacturer ID: b6167f4b4e2441a29319bc6f276089d0
Protocol Type: LWM2M
Software Upgrade Protocol : PCP
Firmware Upgrade Protocol : LWM2M

Step 3 In the **Add Service** area, define the service name, properties, and commands. A service can contain properties and/or commands. Configure the properties and commands based on your requirements.



1. Enter **Service Name** using the camelcase naming method, such as waterMeter or battery.


Basic Information

* Service Name: battery

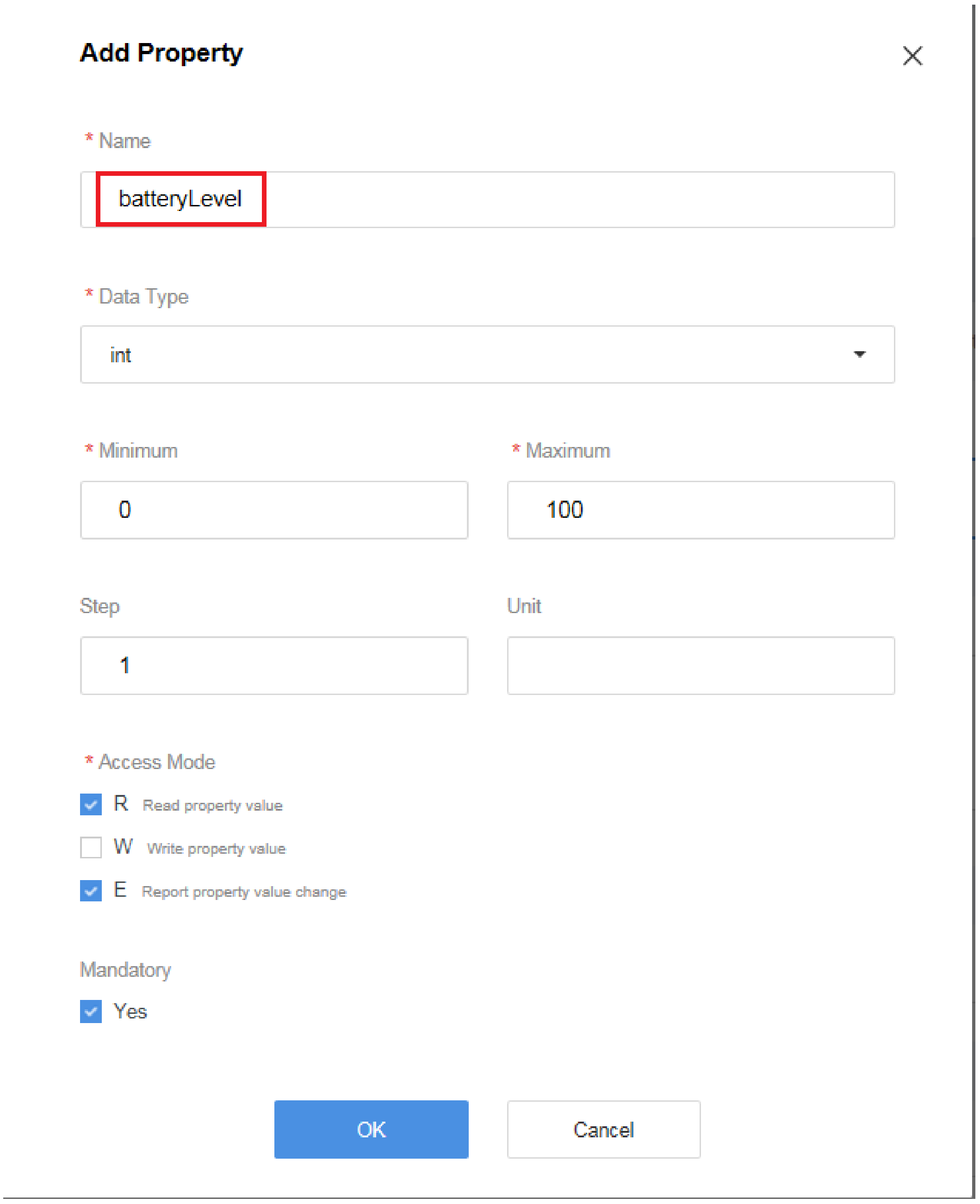
CamelCase naming is recommended, for example, DoorLock and Smoke.

Description :

2. Click **Add** under **Property List**, set the parameters in the dialog box displayed, and click **OK**. For **Name**, the first letter of the first word must be lowercase, and the first letters of subsequent words must be capitalized, for example, batteryLevel or internalTemperature. For other parameters, set them based on your requirements.

   The rules for configuring **Data Type** are as follows:

   – **int**: Select this value if the reported data is an integer or Boolean values.
   – **decimal**: Select this value if the reported data is a decimal.
   – **string**: Select this value if the reported data is a string, enumerated values, or Boolean values. 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.


Add Property

* Name

batteryLevel

* Data Type

int

* Minimum

0

* Maximum

100

Step

1

Unit

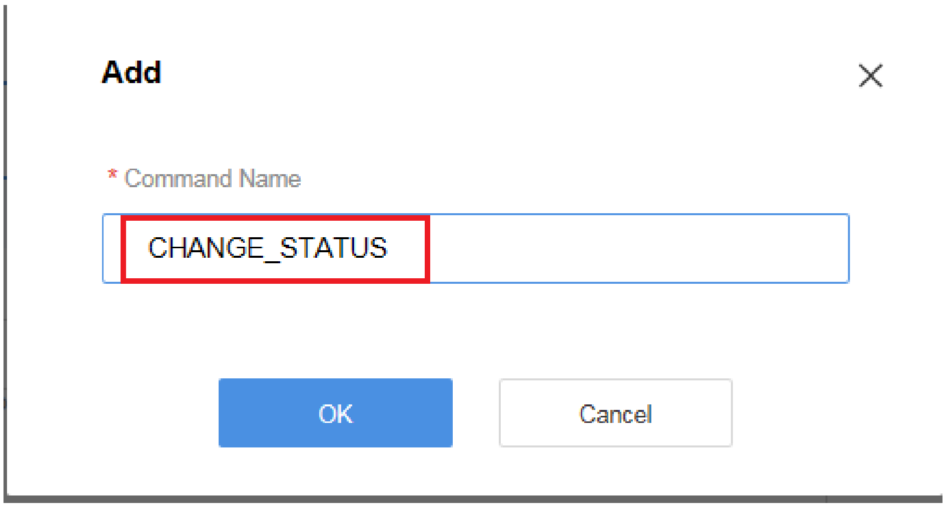
* Access Mode

- [x] R Read property value
- [ ] W Write property value
- [x] E Report property value change

Mandatory

- [x] Yes

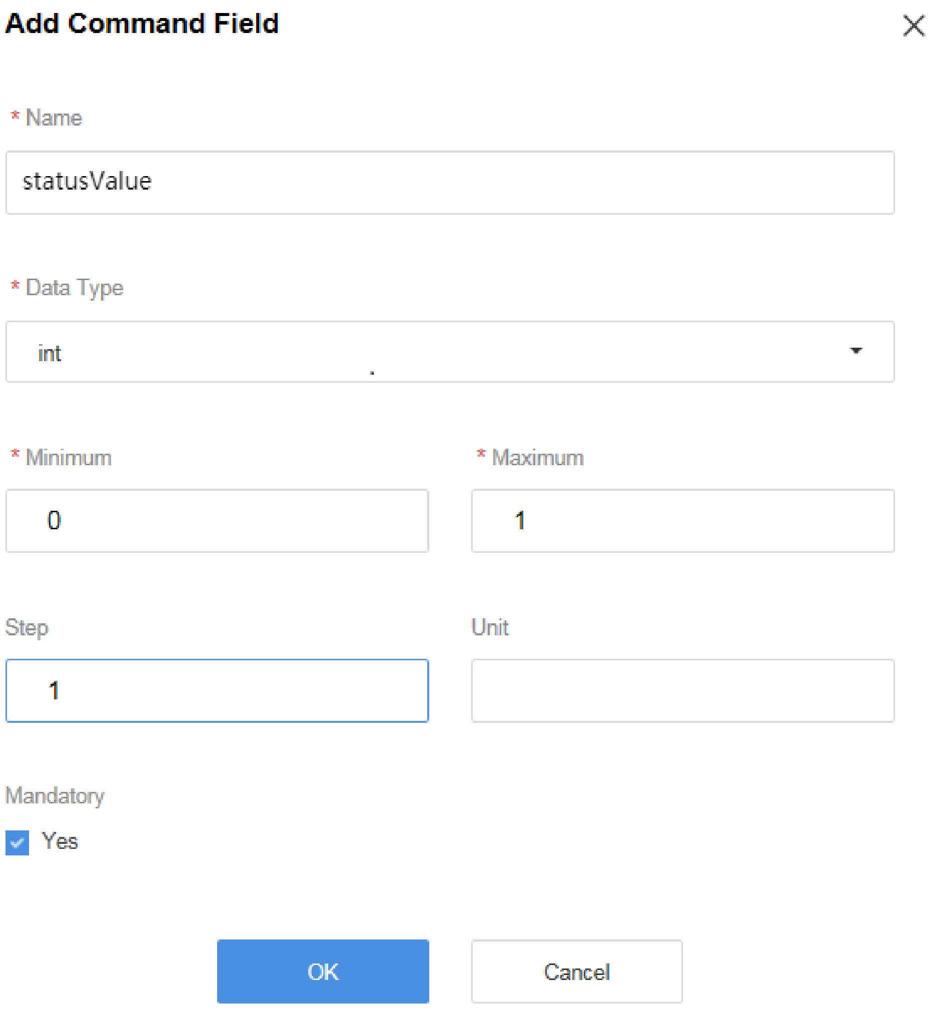
3. Click **Add** under **Command List**. In the dialog box displayed, set **Command Name** and click **OK**. It is recommended that the value of **Command Name** consist of only uppercase letters and underscores (_), for example, DISCOVERY or CHANGE_STATUS.



* Command Name

CHANGE_STATUS

4. Click **Add** under **Command Fields**. In the dialog box displayed, set the parameters and click **OK**. For **Name** of the command field, the first letter of the first word must be lowercase, and the first letters of the subsequent words must be capitalized, for example, statusValue. For other parameters, set them based on your requirements.



* Name: statusValue

* Data Type: int

* Minimum: 0

* Maximum: 1

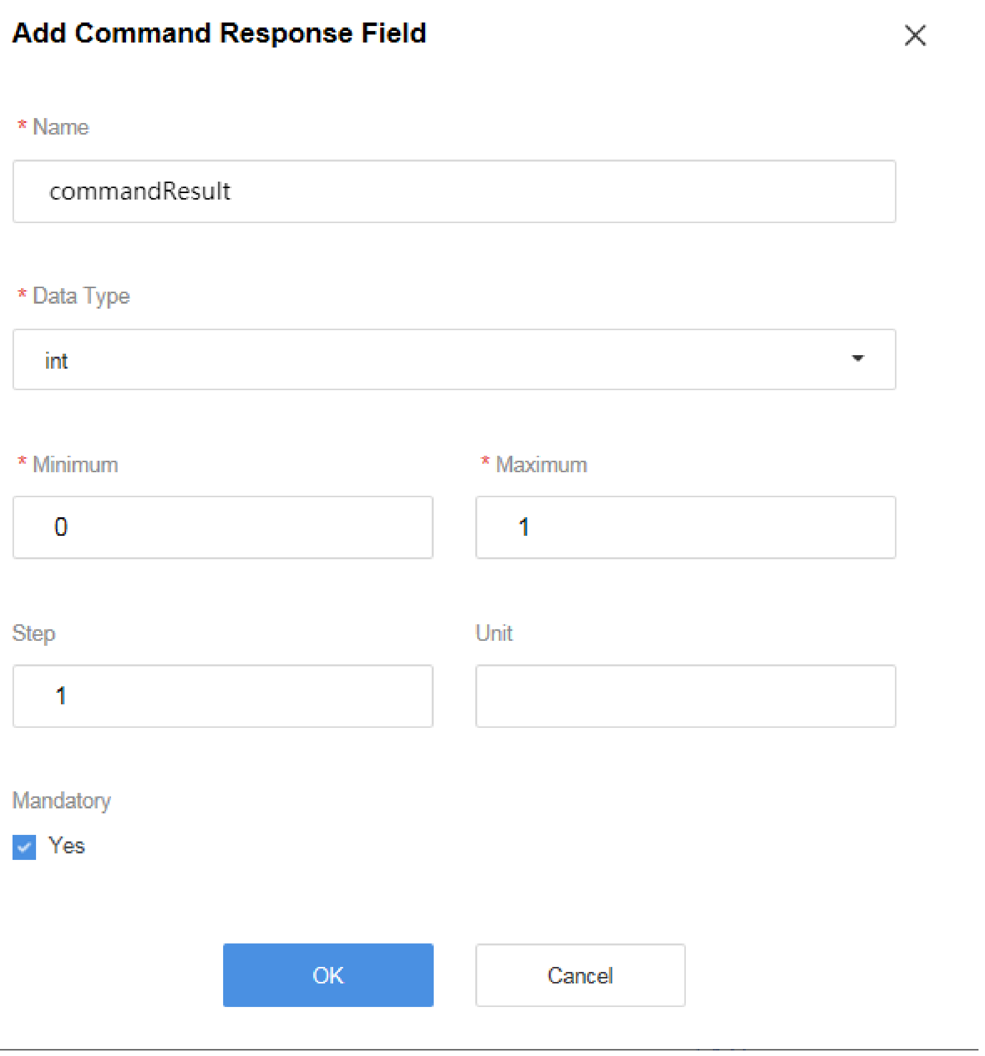
Step: 1

Unit:

Mandatory: ✓ Yes

5. Click **Add** under **Command Response Fields**. In the dialog box displayed, set the parameters and click **OK**. For **Name** of the command response field, the first letter of the first word must be lowercase, and the first letters of the subsequent words must be capitalized, for example, commandResult. For other parameters, set them based on your requirements.

   The command response field is optional. It must be defined only if the device is required to return a command execution result.



Add Command Response Field

* Name: commandResult

* Data Type: int

* Minimum: 0

* Maximum: 1

Step: 1

Unit:

Mandatory: ✓ Yes

----End

1.3.2 Offline Development

1.3.2.1 Profile Writing Guide

A profile is in JSON format.

Identification attributes: include device type, manufacturer, model, and protocol type.

Service list: provides detailed services.

Naming Rules

The profile file must comply with the following naming rules:

- Capitalize device types, service types, and service IDs. Example: **WaterMeter** and **Battery**.
- For the attribute name, uncapitalize the first character in the first world and capitalize the first characters in subsequent words. Example: **batteryLevel** and **internalTemperature**.
- For the order, capitalize all characters, with words separated by underscores. For example: **DISCOVERY** and **CHANGE_COLOR**.
- A device capability profile file (.json file) must be named **devicetype-capability.json**.
- A service capability profile file (.json file) must be named **servicetype-capability.json**.
- The manufacturer ID, manufacturer name, and device model uniquely identify a device. Therefore, their combinations must be unique in different profile files and only English is supported.
- 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**.

NOTE

In some profile file samples, files named **devicetype-display.json** or **servicetype-display.json** may exist. These files are used in some SmartHome scenarios. If they are not involved in the solution communication between you and the IoT platform service provider, these files may not be included in your profile file.

If you need to create a profile file for the SmartHome scenarios, contact the IoT platform support personnel.

Device Profile File

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

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

NOTE

The profile file 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 attributes

Item	Profile Key	Value
Device type	deviceType	WaterMeter
Manufacturer ID	manufacturerId	TestUtf8ManuId
Manufacturer name	manufacturerName	HZYB
Device model	model	NBIoTDevic

Item	Profile Key	Value
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

For details about a complete sample, see **Appendix I: Water Meter Profile Sample**. The service definition can be modified as required. For example, the value ranges or enumerated values of attributes can be modified.

NOTE

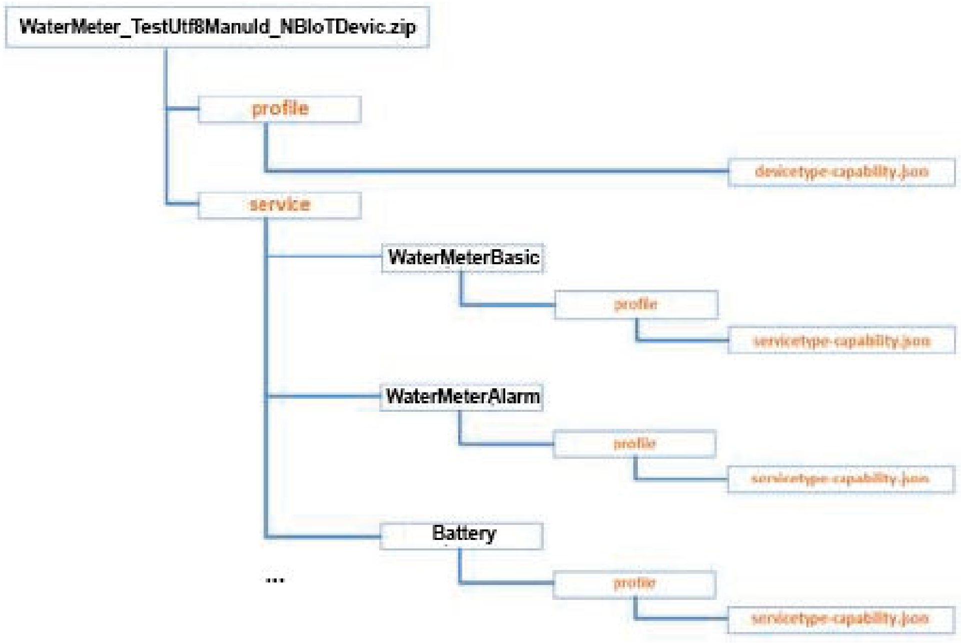
Developers can consult IoT platform support personnel to determine whether the IoT platform supports their own device types. If the device types or service types are supported, developers can obtain the profile file references from the IoT platform support personnel.

A device model is composed of a product type ID and a product ID. For example, if the values of **ProducTypeId** and **ProductId** are **0x0168** and **0x0188**, respectively, the device model is **0168-0188**.

Profile Packaging

After the profile file is completed, package it in the format shown in **Figure 1-1**.

Figure 1-1 Profile file hierarchy


WaterMeter_TestUtf8ManuId_NBIoTDevic.zip
profile
devicetype-capability.json
service
WaterMeterBasic
profile
servicetype-capability.json
WaterMeterAlarm
profile
servicetype-capability.json
Battery
profile
servicetype-capability.json
...

The following requirements must be met for profile packaging:

- The profile file hierarchy must be the same as that shown in **Figure 1-1** 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 in **Figure 1-1** cannot be changed.
- The profile file must be compressed in ZIP format.
- The profile file must be named in the format of deviceType_manufacturerId_model. The values of **deviceType**, **manufacturerId**, and **model** 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",
            "model": "NBIoTDevice",
            "protocolType": "CoAP",
            "deviceType": "WaterMeter",
            "serviceTypeCapabilities": ****
        }
    ]
}
```

- WaterMeterBasic, WaterMeterAlarm, and Battery in **Figure 1-1** are services defined in the **devicetype-capability.json** file.
- The profile file is in JSON format. After the file is edited, you can search for some format verification websites on the Internet to check the validity of the JSON file.

1.3.2.2 Profile Providing Method

You must send the prepared profile file to the Huawei IoT administrator for review. After the approval, the Huawei IoT administrator imports the file to the Huawei IoT lab.

1.3.2.3 Profile Field Description

Device Capabilities

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

```
{
    "devices": [
        {
            "manufacturerId": "TestUtf8ManuId",
            "manufacturerName": "HZYB",
            "model": "NBIoTDevice",
            "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	Contains complete capability information about a device. (The root node cannot be modified.)
	manufacturerId		Mandatory	Identifies the manufacturer of the device.
	manufacturerName		Mandatory	Specifies the manufacturer name of the device. (The value must be in English.)
	model		Mandatory	Specifies the device model. As a type of device may have multiple models, it is recommended that the value contain letters or digits to ensure scalability.
	protocolType		Mandatory	Specifies the protocol used by the device to connect to the IoT platform. For example, the value is **CoAP** for NB-IoT devices.
	deviceType		Mandatory	Specifies the device type.
	omCapability		Optional	Defines the software upgrade, firmware upgrade, and configuration update capabilities of the device. For details, see the description of the omCapability structure in the following. If software or firmware upgrades of the device are not involved, this field can be deleted.
	serviceTypeCapabilities		Mandatory	Describes service capabilities of the device.
		serviceId	Mandatory	Identifies a service. 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	Specifies the service type. The value of this field must be the same as that of **serviceType** in the **servicetype-capability.json** file.
		option	Mandatory	Specifies the service type. The value can be **Master**, **Mandatory**, or **Optional**. This field is not a functional field but a descriptive one.

Description of the omCapability structure

Field	Sub-field	Mandatory or Optional	Description
upgradeCapability		Optional	Specifies software upgrade capabilities of a device.
	supportUpgrade	Optional	**true**: The device supports software upgrades. **false**: The device does not support software upgrades.
	upgradeProtocolType	Optional	Specifies the protocol type used by the device for upgrades. It is different from **protocolType** of the device. For example, the software upgrade protocol of CoAP devices is PCP.
fwUpgradeCapability		Optional	Specifies firmware upgrade capabilities of the device.
	supportUpgrade	Optional	**true**: The device supports firmware upgrades. **false**: The device does not support firmware upgrades.
	upgradeProtocolType	Optional	Specifies the protocol type used by the device for upgrades. It is different from **protocolType** of the device. Currently, the IoT platform supports only firmware upgrade of LWM2M devices.
configCapability		Optional	Specifies 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	Specifies the default device configuration information (in JSON format). The specific configuration information is defined by the manufacturers. The IoT platform only stores the information for delivery and does not parse the meaning of the configuration fields.

Service Capabilities

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": "M",
                    "unit": "L",
                    "enumList": null
                },
                {
                    "propertyName": "timeOfReading",
                    "dataType": "string",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "M",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "internalTemperature",
                    "dataType": "int",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "M",
                    "unit": "0.01°C",
                    "enumList": null
                },
                {
                    "propertyName": "dailyFlow",
                    "dataType": "int",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "M",
                    "unit": "L",
                    "enumList": null
                },
                {
                    "propertyName": "dailyReverseFlow",
                    "dataType": "int",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "M",
                    "unit": "L",
                    "enumList": null
                },
                {
                    "propertyName": "peakFlowRate",
                    "dataType": "int",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "M",
                    "unit": "L/H",
                    "enumList": null
                },
                {
                    "propertyName": "peakFlowRateTime",
                    "dataType": "string",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "M",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "intervalFlow",
                    "dataType": "array",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "M",
                    "unit": "L",
                    "enumList": null
                },
                {
                    "propertyName": "pressure",
                    "dataType": "array",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "O",
                    "unit": "kPa",
                    "enumList": null
                },
                {
                    "propertyName": "temperature",
                    "dataType": "array",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "M",
                    "unit": "0.01°C",
                    "enumList": null
                },
                {
                    "propertyName": "vibration",
                    "dataType": "array",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "M",
                    "unit": "0.01g",
                    "enumList": null
                }
            ]
        }
    ]
}
```

The fields are described as follows:

Field	Sub-field				Mandatory or Optional	Description
services					Mandatory	Contains complete information about a service. (The root node cannot be modified.)
	serviceType				Mandatory	Specifies the service type. The value of this field must be the same as that of **serviceType** in the **devicetype-capability.json** file.
	description				Mandatory	Provides description about the service. This field is not a functional field but a descriptive one. It can be set to null.
	commands				Mandatory	Specifies a parameter that a device can run. If the service has no commands, set the value to **null**.
		commandName			Mandatory	Specifies the name of a command. The command name and parameters together form a complete command.
		paras			Mandatory	Specifies parameters contained in a command.
			paraName		Mandatory	Specifies the name of a parameter in the command.
			dataType		Mandatory	Specifies the data type of a command parameter. 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 yyyyMMddTHHmmssZ, for example, 20151212T121212Z. ● **jsonObject**: The value is in customized JSON structure, which is not parsed by the IoT platform and is transparently transmitted only.

Field	Sub-field				Mandatory or Optional	Description
			required		Mandatory	Specifies whether the command is mandatory. The value can be **true** or **false**. The default value is **false** (optional). This field is not a functional field but a descriptive one.
			min		Mandatory	Specifies the minimum value. This parameter is valid only when **dataType** is set to **int** or **decimal**.
			max		Mandatory	Specifies the maximum value. This parameter is valid only when **dataType** is set to **int** or **decimal**.
			step		Mandatory	Specifies the step. This field is not used. Set it to **0**.
			maxLength		Mandatory	Specifies the character string length. This field is valid only when **dataType** is **string**, **string list**, or **DateTime**.
			unit		Mandatory	Specifies the unit. The value is determined by the parameter, for example: Temperature unit: C or K Percentage unit: % Pressure unit: Pa or kPa
			enumList		Mandatory	Specifies a 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	Specifies responses to command execution.
			responseName		Mandatory	You can add _RSP to the end of **commandName** in the command corresponding to **responses**.

Field	Sub-field				Mandatory or Optional	Description
			paras		Mandatory	Specifies parameters contained in a response.
				paraName	Mandatory	Specifies the name of a parameter in the command.
				dataType	Mandatory	Specifies the 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 yyyyMMddTHHmmssZ, for example, 20151212T121212Z. ● **jsonObject**: The value is in customized JSON structure, which is not parsed by the IoT platform and is transparently transmitted only.
				required	Mandatory	Specifies whether the command response is mandatory. The value can be **true** or **false**. The default value is **false** (optional). This field is not a functional field but a descriptive one.
				min	Mandatory	Specifies the minimum value. This field is valid only when **dataType** is **int** or **decimal**. The value of a field of the **int** or **decimal** type must be greater than or equal to the value of **min**.
				max	Mandatory	Specifies the maximum value. This field is valid only when **dataType** is **int** or **decimal**. The value of a field of the **int** or **decimal** type must be less than or equal to the value of **max**.
				step	Mandatory	Specifies the step. This field is not used. Set it to **0**.

Field	Sub-field				Mandatory or Optional	Description
				maxLength	Mandatory	Specifies the character string length. This field is valid only when **dataType** is **string**, **string list**, or **DateTime**.
				unit	Mandatory	Specifies the unit. The value is determined by the parameter, for example: Temperature unit: C or K Percentage unit: % Pressure unit: Pa or kPa
				enumList	Mandatory	Specifies a 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	Describes reported data. Each sub-node indicates an attribute.
		propertyName			Mandatory	Specifies the attribute name.
		dataType			Mandatory	Specifies the 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 yyyyMMddTHHmmssZ, for example, 20151212T121212Z. ● **jsonObject**: The value is in customized JSON structure, which is not parsed by the IoT platform and is transparently transmitted only.

Field	Sub-field				Mandatory or Optional	Description
		required			Mandatory	Specifies whether an attribute is mandatory. The value can be **true** or **false**. The default value is **false**, which indicates that the attribute is optional. This field is not a functional field but a descriptive one.
		min			Mandatory	Specifies the minimum value. This field is valid only when **dataType** is **int** or **decimal**. The value of a field of the **int** or **decimal** type must be greater than or equal to the value of **min**.
		max			Mandatory	Specifies the maximum value. This field is valid only when **dataType** is **int** or **decimal**. The value of a field of the **int** or **decimal** type must be less than or equal to the value of **max**.
		step			Mandatory	Specifies the step. This field is not used. Set it to **0**.
		method			Mandatory	Specifies the access mode. ● **R**: readable ● **W**: writable ● **E**: subscription Value: R, RW, RE, RWE, or null
		unit			Mandatory	Specifies the unit. The value is determined by the parameter, for example: Temperature unit: C or K Percentage unit: % Pressure unit: Pa or kPa
		maxLength			Mandatory	Specifies the character string length. This field is valid only when **dataType** is **string**, **string list**, or **DateTime**.

Field	Sub-field				Mandatory or Optional	Description
		enumList			Mandatory	Specifies a 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.

1.3.3 Reference

1.3.3.1 Product Model Sample

Appendix I: Water Meter Profile Sample

A water meter profile sample contains six files, whose names and content are described as follows:

1. **devicetype-capability.json**

```
{
    "devices": [
        {
            "manufacturerId": "TestUtf8ManuId",
            "manufacturerName": "HZYB",
            "model":  "NBIoTDevice",
            "protocolType": "CoAP",
            "deviceType": "WaterMeter",
            "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"
                }
            ]
        }
    ]
}
```

2. **servicetype-capability.json** (Battery)

```
{
    "services": [
        {
            "serviceType": "Battery",
            "description": "Battery",
            "commands": null,
            "properties": [
                {
                    "propertyName": "batteryLevel",
                    "dataType": "int",
                    "required": true,
                    "min": 0,
                    "max": 100,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": "%",
                    "enumList": null
                },
                {
                    "propertyName": "batteryThreshold",
                    "dataType": "int",
                    "required": false,
                    "min": 0,
                    "max": 100,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": "%",
                    "enumList": null
                },
                {
                    "propertyName": "batteryStatus",
                    "dataType": "int",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": null,
                    "enumList": [
                        0,
                        1,
                        2,
                        3,
                        4,
                        5,
                        6
                    ]
                }
            ]
        }
    ]
}
```

3. **servicetype-capability.json** (ConnectivityMonitoring)

```
{
    "services": [
        {
            "serviceType": "Connectivity",
            "description": "Connectivity",
            "commands": null,
            "properties": [
                {
                    "propertyName": "signalStrength",
                    "dataType": "int",
                    "required": true,
                    "min": -110,
                    "max": -48,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": "dbm",
                    "enumList": null
                },
                {
                    "propertyName": "linkQuality",
                    "dataType": "int",
                    "required": false,
                    "min": -110,
                    "max": -48,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": "dbm",
                    "enumList": null
                },
                {
                    "propertyName": "cellId",
                    "dataType": "int",
                    "required": false,
                    "min": 0,
                    "max": 268435455,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": null,
                    "enumList": null
                }
            ]
        }
    ]
}
```

4. **servicetype-capability.json** (DeliverySchedule)

```
{
    "services": [
        {
            "serviceType": "DeliverySchedule",
            "description": "DeliverySchedule",
            "commands": null,
            "properties": [
                {
                    "propertyName": "startTime",
                    "dataType": "int",
                    "required": true,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RW",
                    "unit": "sec",
                    "enumList": null
                },
                {
                    "propertyName": "UTCOffset",
                    "dataType": "string",
                    "required": true,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RW",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "frequency",
                    "dataType": "int",
                    "required": true,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RW",
                    "unit": "sec",
                    "enumList": null
                },
                {
                    "propertyName": "randomisedDeliveryWindow",
                    "dataType": "int",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RW",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "retries",
                    "dataType": "int",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RW",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "retryPeriod",
                    "dataType": "int",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RW",
                    "unit": null,
                    "enumList": null
                }
            ]
        }
    ]
}
```

5. **servicetype-capability.json** (WaterMeterAlarm)

```
{
    "services": [
        {
            "serviceType": "WaterMeterAlarm",
            "description": "WaterMeterAlarm",
            "commands": null,
            "properties": [
                {
                    "propertyName": "lowFlowAlarm",
                    "dataType": "int",
                    "required": true,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "highFlowAlarm",
                    "dataType": "int",
                    "required": true,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "tamperAlarm",
                    "dataType": "int",
                    "required": true,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "lowBatteryAlarm",
                    "dataType": "int",
                    "required": true,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "batteryRunOutAlarm",
                    "dataType": "int",
                    "required": true,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "highInternalTemperature",
                    "dataType": "int",
                    "required": true,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "reverseFlowAlarm",
                    "dataType": "int",
                    "required": true,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "highPressureAlarm",
                    "dataType": "int",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "lowPressureAlarm",
                    "dataType": "int",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "highTemperatureAlarm",
                    "dataType": "int",
                    "required": true,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "lowTemperatureAlarm",
                    "dataType": "int",
                    "required": true,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "innerErrorAlarm",
                    "dataType": "int",
                    "required": true,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "storageFault",
                    "dataType": "int",
                    "required": true,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "waterTempratureSensorFault",
                    "dataType": "int",
                    "required": true,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "innerTempratureSensorFault",
                    "dataType": "int",
                    "required": true,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "pressureSensorFault",
                    "dataType": "int",
                    "required": true,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "vibrationSensorFault",
                    "dataType": "int",
                    "required": true,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "strayCurrent",
                    "dataType": "int",
                    "required": true,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "RE",
                    "unit": null,
                    "enumList": null
                }
            ]
        }
    ]
}
```

6. **servicetype-capability.json** (WaterMeterBasic)

```
{
    "services": [
        {
            "serviceType": "WaterMeterBasic",
            "description": "WaterMeterBasic",
            "commands": 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
                },
                {
                    "propertyName": "internalTemperature",
                    "dataType": "int",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "W",
                    "unit": "0.01°C",
                    "enumList": null
                },
                {
                    "propertyName": "dailyFlow",
                    "dataType": "int",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "W",
                    "unit": "L",
                    "enumList": null
                },
                {
                    "propertyName": "dailyReverseFlow",
                    "dataType": "int",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "W",
                    "unit": "L",
                    "enumList": null
                },
                {
                    "propertyName": "peakFlowRate",
                    "dataType": "int",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "W",
                    "unit": "L/H",
                    "enumList": null
                },
                {
                    "propertyName": "peakFlowRateTime",
                    "dataType": "string",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "W",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "intervalFlow",
                    "dataType": "array",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "W",
                    "unit": "L",
                    "enumList": null
                },
                {
                    "propertyName": "pressure",
                    "dataType": "array",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "W",
                    "unit": "kPa",
                    "enumList": null
                },
                {
                    "propertyName": "temperature",
                    "dataType": "array",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "W",
                    "unit": "0.01°C",
                    "enumList": null
                },
                {
                    "propertyName": "vibration",
                    "dataType": "array",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "W",
                    "unit": "0.01g",
                    "enumList": null
                }
            ]
        }
    ]
}
```

1.3.3.2 Fields in the Profile Sample

Device Capabilities

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

```
{
    "devices": [
        {
            "manufacturerId": "TestUtf8ManuId",
            "manufacturerName": "HZYB",
            "model": "NBIoTDevice",
            "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		Mandatory	Manufacturer ID of the device.
	manufacturerName		Mandatory	Manufacturer name of the device. (The value must be in English.)
	model		Mandatory	Device model. As a type of device may have multiple models, it is recommended that the value contain letters or digits to ensure scalability.
	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.

Field	Sub-field		Mandatory or Optional	Description
		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

Field	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.

Field	Sub-field	Mandatory or Optional	Description
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 Capabilities

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
                },
                {
                    "propertyName": "internalTemperature",
                    "dataType": "int",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "W",
                    "unit": "0.01°C",
                    "enumList": null
                },
                {
                    "propertyName": "dailyFlow",
                    "dataType": "int",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "W",
                    "unit": "L",
                    "enumList": null
                },
                {
                    "propertyName": "dailyReverseFlow",
                    "dataType": "int",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "W",
                    "unit": "L",
                    "enumList": null
                },
                {
                    "propertyName": "peakFlowRate",
                    "dataType": "int",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "W",
                    "unit": "L/H",
                    "enumList": null
                },
                {
                    "propertyName": "peakFlowRateTime",
                    "dataType": "string",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "W",
                    "unit": null,
                    "enumList": null
                },
                {
                    "propertyName": "intervalFlow",
                    "dataType": "array",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "W",
                    "unit": "L",
                    "enumList": null
                },
                {
                    "propertyName": "pressure",
                    "dataType": "array",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "W",
                    "unit": "kPa",
                    "enumList": null
                },
                {
                    "propertyName": "temperature",
                    "dataType": "array",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "W",
                    "unit": "0.01°C",
                    "enumList": null
                },
                {
                    "propertyName": "vibration",
                    "dataType": "array",
                    "required": false,
                    "min": 0,
                    "max": 0,
                    "step": 1,
                    "maxLength": 0,
                    "method": "W",
                    "unit": "0.01g",
                    "enumList": null
                }
            ]
        }
    ]
}
```

The fields are described as follows:

Field	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.

Field	Sub-field				Mandatory or Optional	Description
			dataType		Mandatory	Data type of the parameter in the command. 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 but 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**.
			unit		Mandatory	Unit. The value is determined by the parameter, for example: Temperature unit: C or K Percentage unit: % Pressure unit: Pa or kPa

Field	Sub-field				Mandatory or Optional	Description
			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.
				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 but 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.

Field	Sub-field				Mandatory or Optional	Description
				min	Mandatory	Minimum value. This field is valid only when **dataType** is set to **int** or **decimal**. The value of a field of the **int** or **decimal** type must be greater than or equal to the value of **min**.
				max	Mandatory	Maximum value. This field is valid only when **dataType** is set to **int** or **decimal**. The value of a field of the **int** or **decimal** type must be less than or equal to the value of **max**.
				step	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**.
				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.

Field	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 but is transparently transmitted only.
		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 of a field of the **int** or **decimal** type 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 of a field of the **int** or **decimal** type 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**: readable ● **W**: writable ● **E**: subscription Value: R, RW, RE, RWE, or null

Field	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
		maxLength			Mandatory	Character string length. This field is valid only when **dataType** is set to **string**, **string list**, or **DateTime**.
		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.

1.4 Developing a Codec

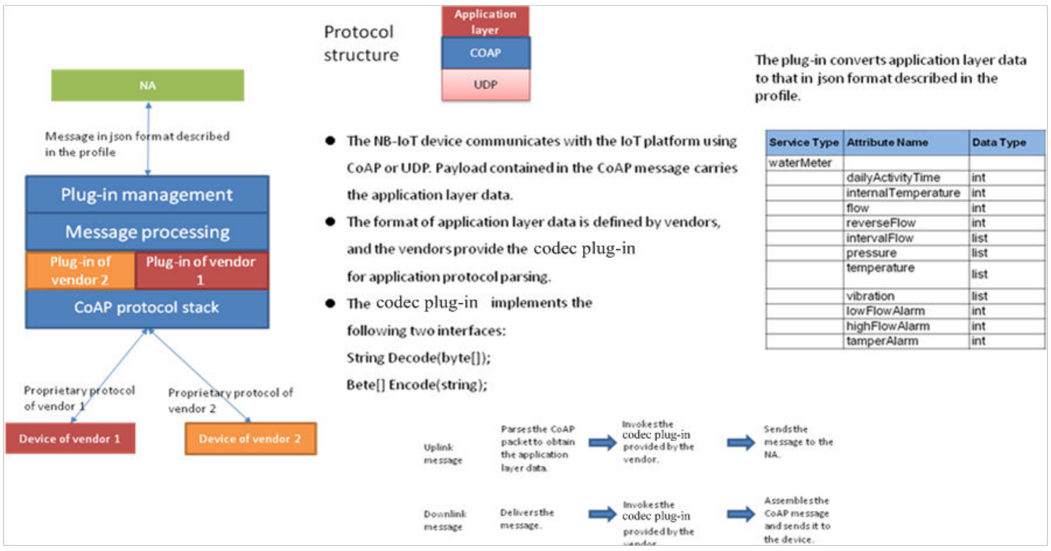
1.4.1 Development Guide

Overview

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 IoT platform communicates with NAs by sending data in JSON format. Therefore, codec development is required for the IoT platform to convert data in binary and JSON formats.

Figure 1-2 Codec conversion


Protocol structure
Application layer
COAP
UDP
NA
Message in json format described in the profile
Plug-in management
Message processing
Plug-in of vendor 2
Plug-in of vendor 1
CoAP protocol stack
Proprietary protocol of vendor 1
Proprietary protocol of vendor 2
Device of vendor 1
Device of vendor 2
Uplink message
Parses the CoAP packet to obtain the application layer data.
Invokes the codec plug-in provided by the vendor.
Sends the message to the NA.
Downlink message
Delivers the message.
Invokes the codec plug-in provided by the
Assembles the CoAP message and sends it to the device.

The plug-in converts application layer data to that in json format described in the profile.

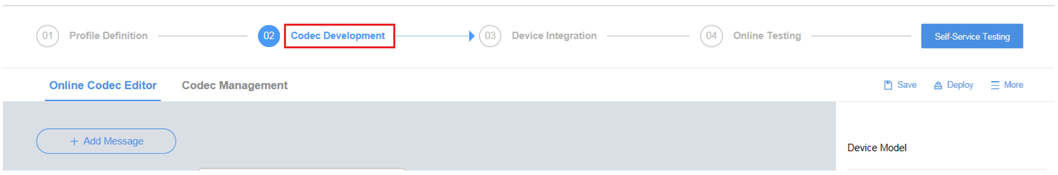
- The NB-IoT device communicates with the IoT platform using CoAP or UDP. Payload contained in the CoAP message carries the application layer data.
- The format of application layer data is defined by vendors, and the vendors provide the codec plug-in for application protocol parsing.
- The codec plug-in implements the following two interfaces:
  String Decode(byte[]);
  Bete[] Encode(string);

Service Type	Attribute Name	Data Type
waterMeter		
	dailyActivityTime	int
	internalTemperature	int
	flow	int
	reverseFlow	int
	intervalFlow	list
	pressure	list
	temperature	list
	vibration	list
	lowFlowAlarm	int
	highFlowAlarm	int
	tamperAlarm	int

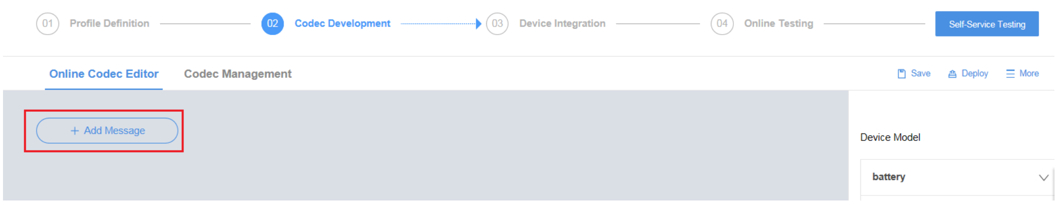
Procedure

If you have used a preset template when **creating a project and product**, you can directly use or modify the codecs contained in the template. If a customized product is created, you must develop your codec.

Step 1 In the product development space, click **Codec Development**.

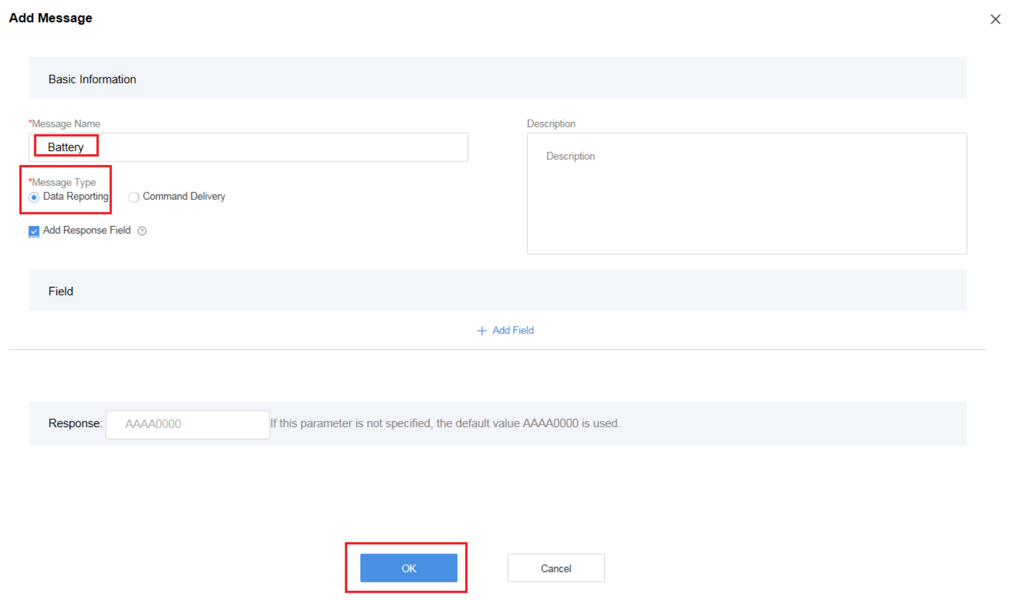


Step 2 In the **Online Codec Editor** area, click **Add Message**.

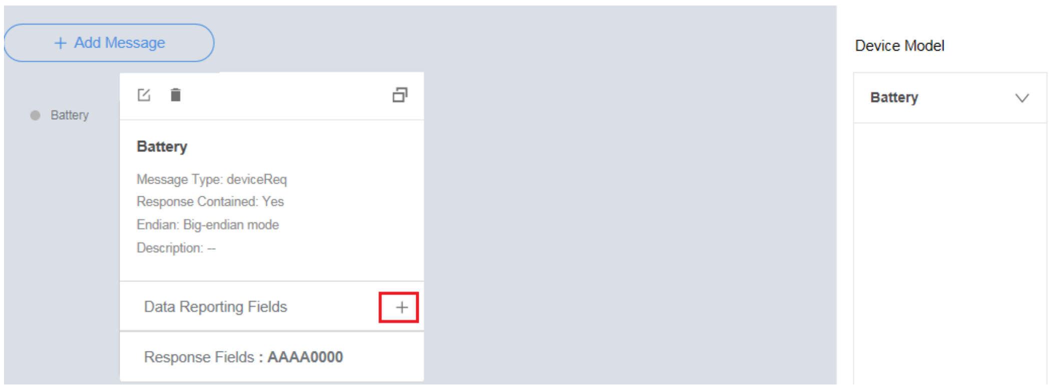


Step 3 In the **Add Message** dialog box, specify **Message Name**, set **Message Type** to **Data Reporting**, and click **OK**.

- If the IoT platform is required to return an ACK message after receiving data reported by the device, select **Add Response Field**. The data carried in the ACK message is the value of **Response**. The default value is **AAAA0000**.
- **Message Name** can contain only letters, digits, underscores (_), and dollar signs ($) and cannot start with a digit.

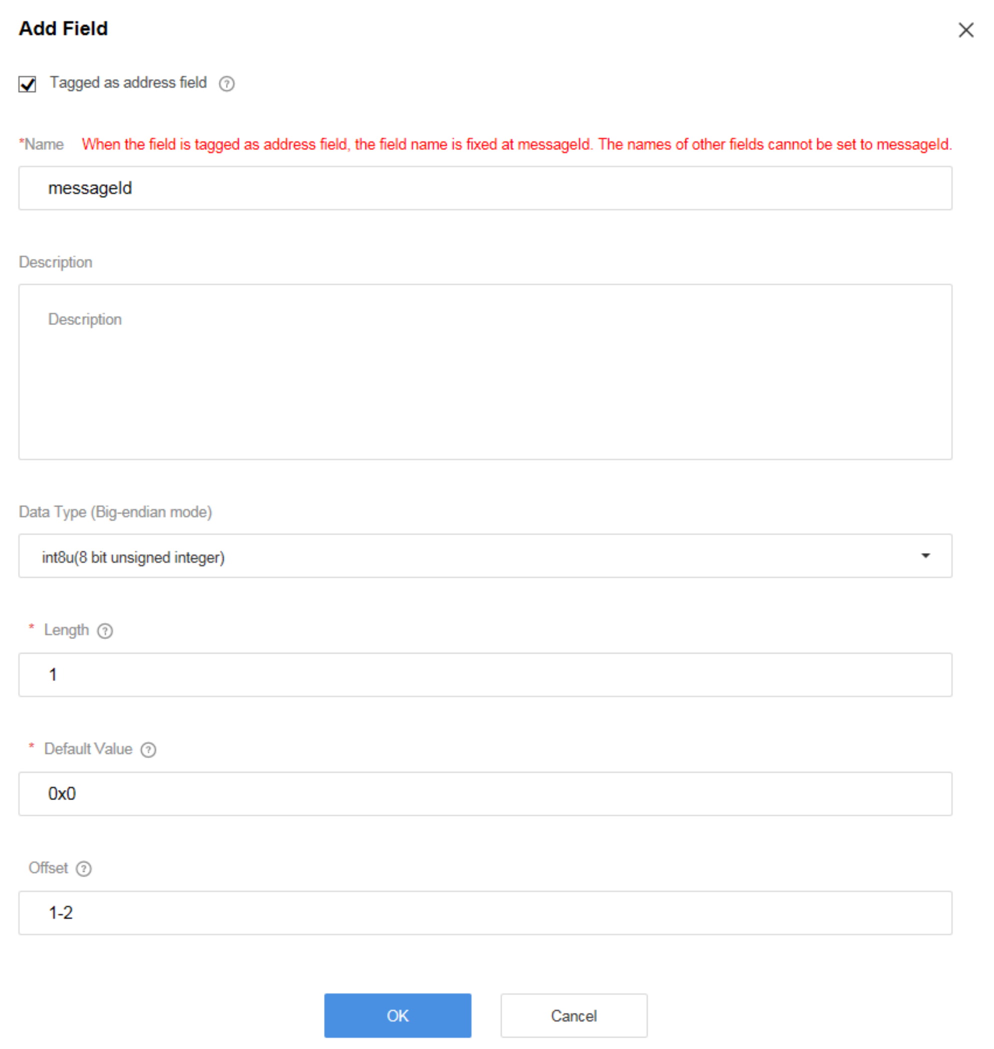


Step 4 Click + next to **Data Reporting Fields**.



Step 5 In the **Add Field** dialog box, select **Tagged as address field**. Other parameters are set automatically. Click **OK**.

When messages of the same type are created, such as two data reporting messages, **Tagged as address field** must be selected and this field in every such message must be in the same place on the field list. A command response can be regarded as a type of data reporting message. Therefore, if a command response exists, **messageId** must be added to the data reporting message.


Add Field

☑ Tagged as address field

*Name When the field is tagged as address field, the field name is fixed at messageId. The names of other fields cannot be set to messageId.

messageId

Description

Description

Data Type (Big-endian mode)

int8u(8 bit unsigned integer)

* Length

1

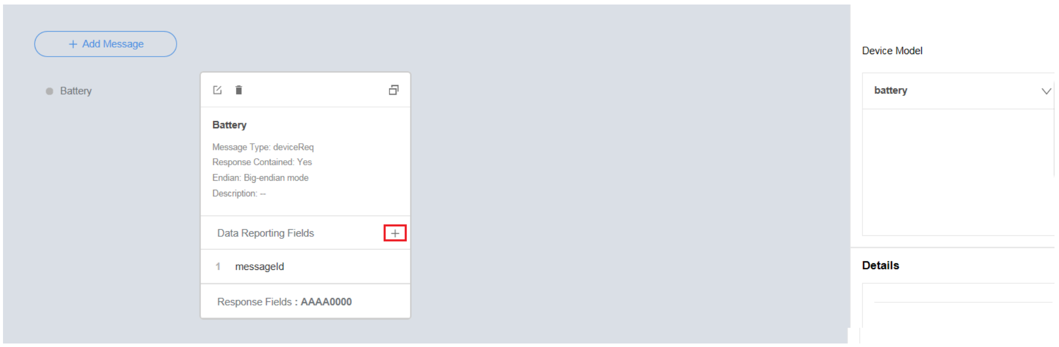
* Default Value

0x0

Offset

1-2

Step 6 Click + next to **Data Reporting Fields**.



Battery

Message Type: deviceReq
Response Contained: Yes
Endian: Big-endian mode
Description: --

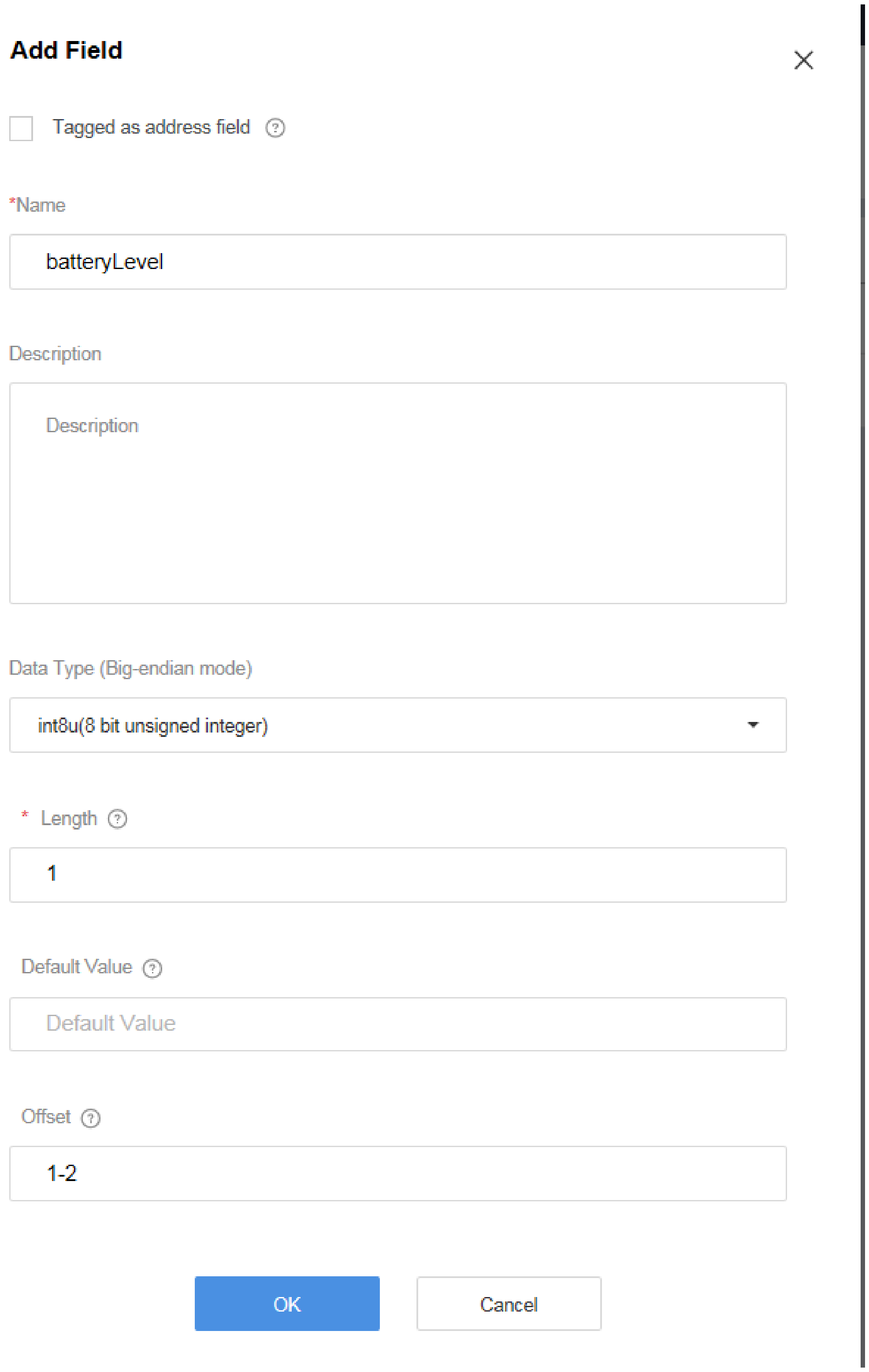
Data Reporting Fields +

1 messageId

Response Fields : AAAA0000

Step 7 In the **Add Field** dialog box, set the parameters and click **OK**.

- **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 profile file.


Add Field

Tagged as address field

*Name

batteryLevel

Description

Description

Data Type (Big-endian mode)

int8u(8 bit unsigned integer)

* Length

1

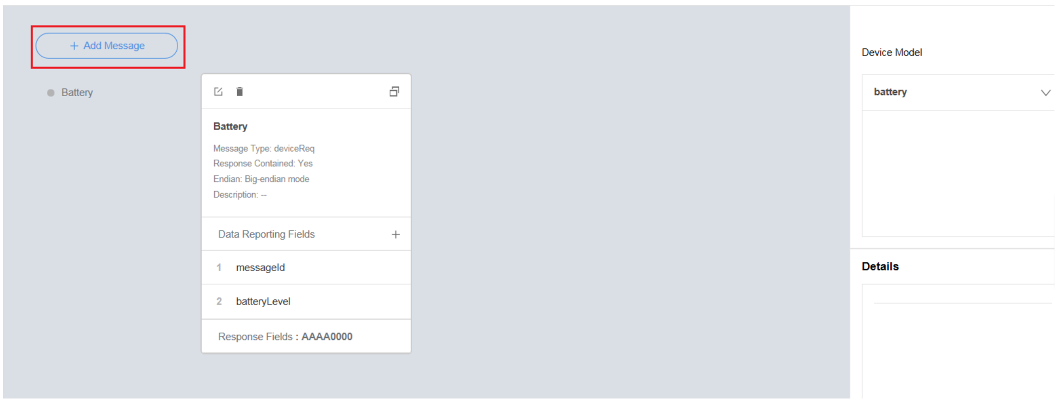
Default Value

Default Value

Offset

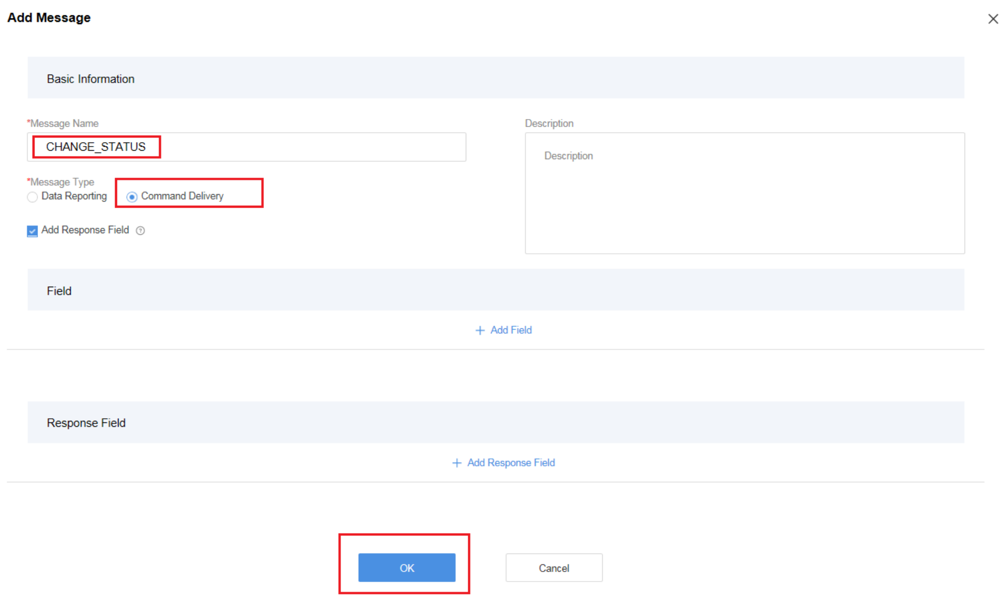
1-2

Step 8 In the **Online Codec Editor** area, click **Add Message**.



Step 9 In the **Add Message** dialog box, specify **Message Name**, set **Message Type** to **Command Delivery**, and click **OK**.

- If the device is required to return the command execution result, select **Add Response Field**. After the check box is selected:
  - The address field must be defined in both the data reporting message and the command response, and this field in the two messages must be in the same place on the field list, so that the codec can distinguish the data reporting message from the command response.
  - The response ID field must be defined in the command delivery message and the command response, and this field in the two messages must be in the same place on the field list, so that the codec can associate the command delivery message with the corresponding command response.
- **Message Name** can contain only letters, digits, underscores (_), and dollar signs ($) and cannot start with a digit.


Add Message

Basic Information

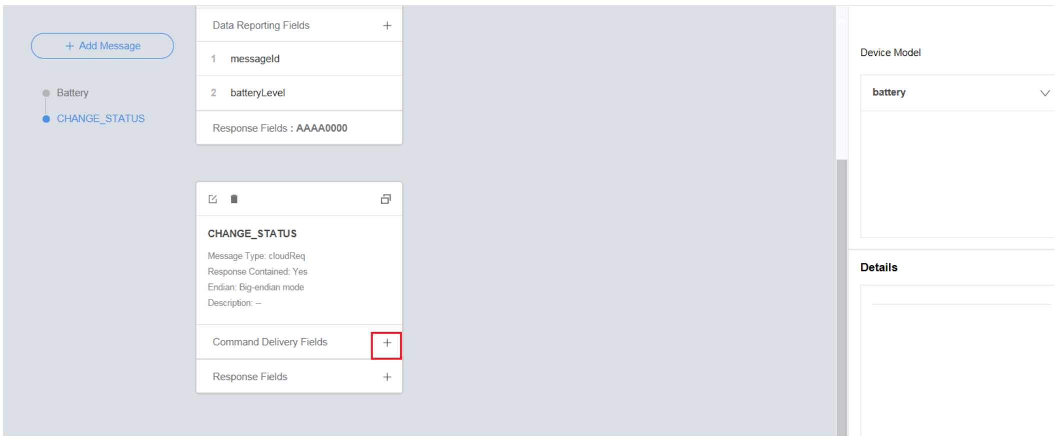
*Message Name: CHANGE_STATUS

*Message Type: Data Reporting [✓ Command Delivery]

[✓] Add Response Field

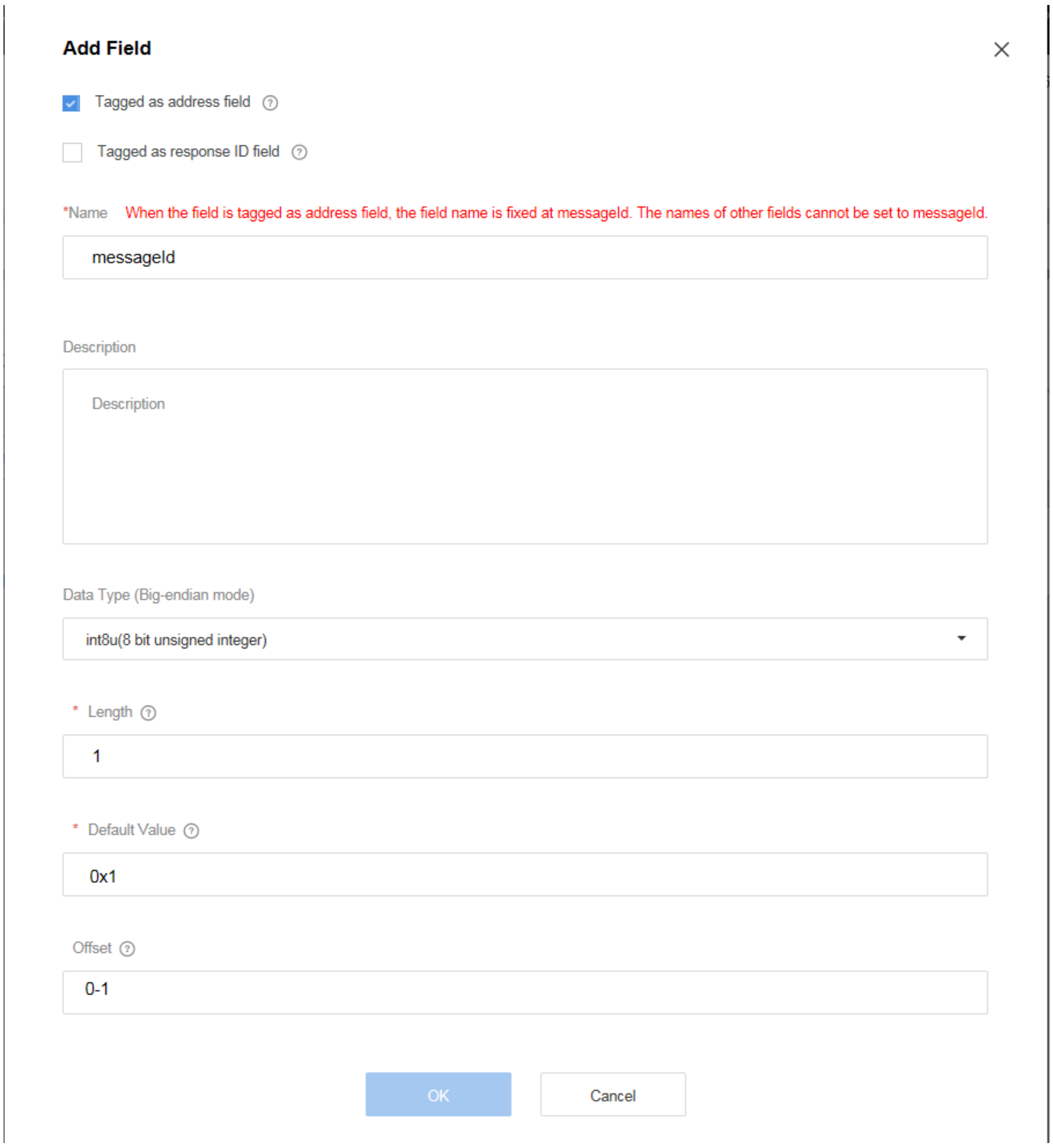
Description: Description

Step 10 Click + next to **Command Delivery Fields**.



Step 11 In the **Add Field** dialog box, select **Tagged as address field**. Other parameters are set automatically. Click **OK**.

When messages of the same type are created, such as two command delivery messages, **Tagged as address field** must be selected and this field in every such message must be in the same place on the field list. A data reporting response can be regarded as a type of command delivery message. Therefore, if a data reporting response exists, **messageId** must be added to the command delivery message.


Add Field

☑ Tagged as address field

☐ Tagged as response ID field

*Name When the field is tagged as address field, the field name is fixed at messageId. The names of other fields cannot be set to messageId.

messageId

Description

Description

Data Type (Big-endian mode)

int8u(8 bit unsigned integer)

* Length

1

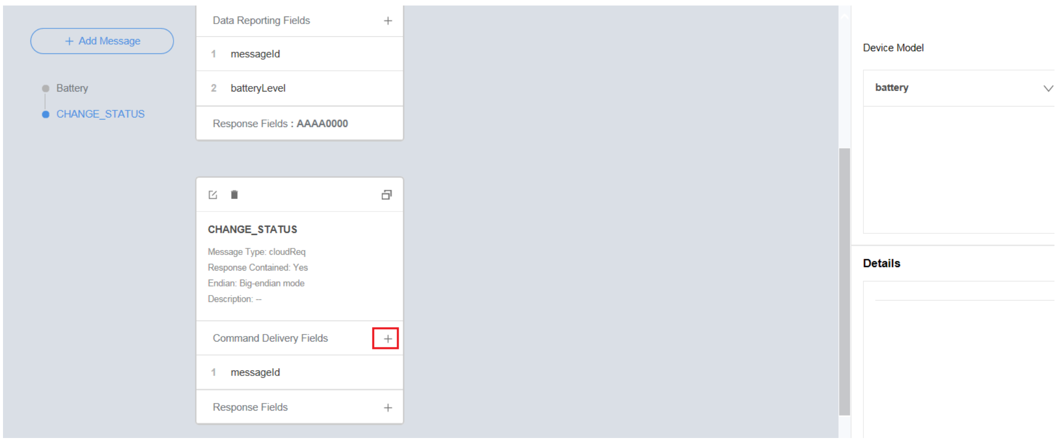
* Default Value

0x1

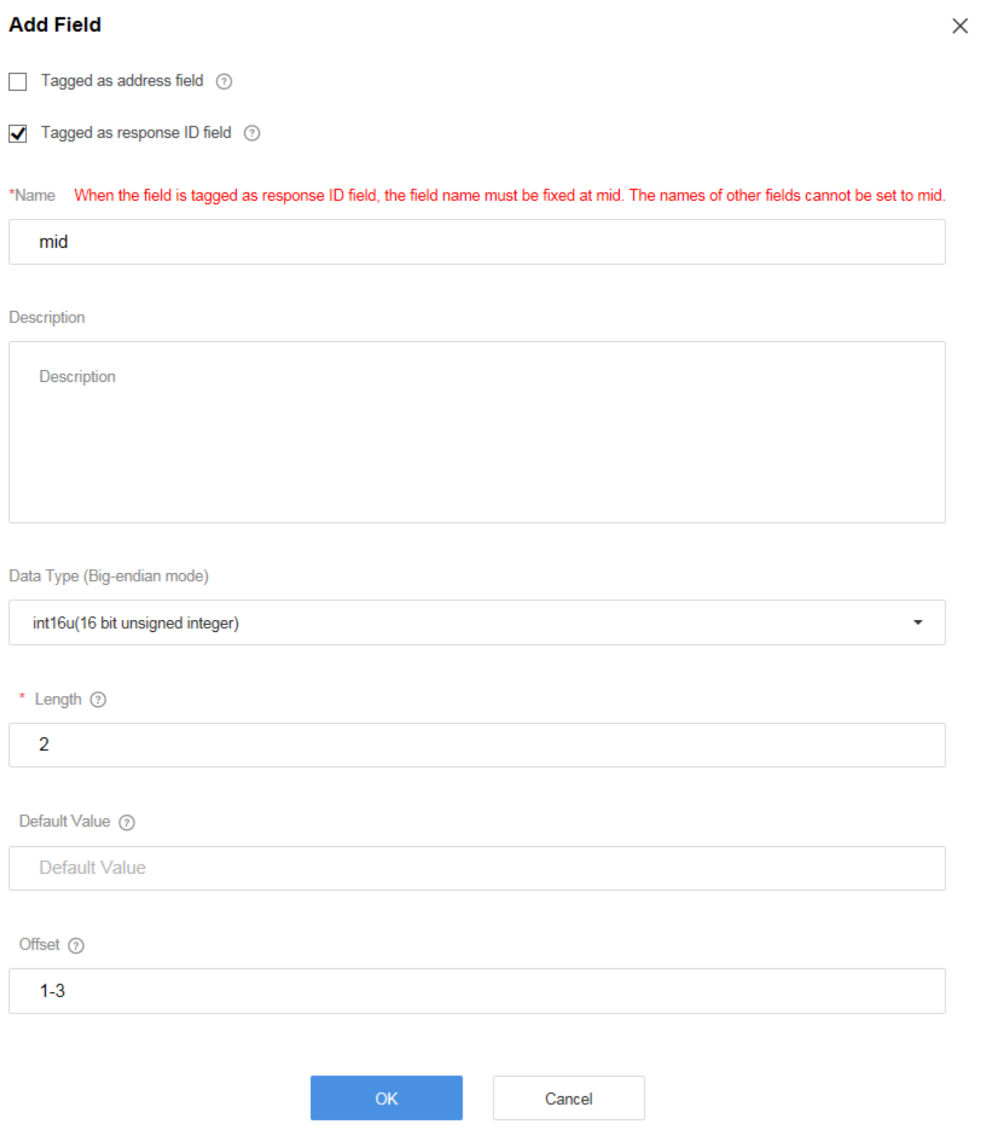
Offset

0-1

Step 12 Click + next to **Command Delivery Fields**.



Step 13 In the **Add Field** dialog box, select **Tagged as response ID field**. Other parameters are set automatically. Click **OK**.



Add Field

☐ Tagged as address field

☑ Tagged as response ID field

*Name When the field is tagged as response ID field, the field name must be fixed at mid. The names of other fields cannot be set to mid.

mid

Description

Description

Data Type (Big-endian mode)

int16u(16 bit unsigned integer)

* Length

2

Default Value

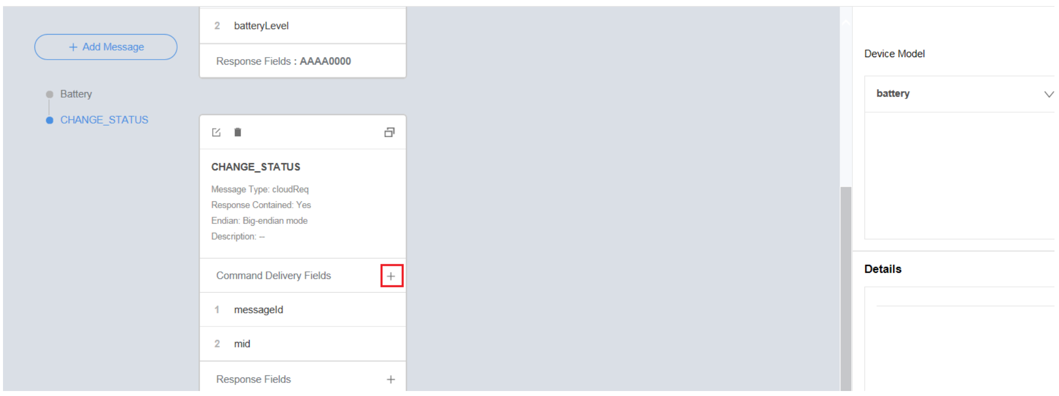
Default Value

Offset

1-3

OK Cancel

Step 14 Click + next to **Command Delivery Fields**.



2 batteryLevel

Response Fields : AAAA0000

CHANGE_STATUS

Message Type: cloudReq
Response Contained: Yes
Endian: Big-endian mode
Description: --

Command Delivery Fields +

1 messageId

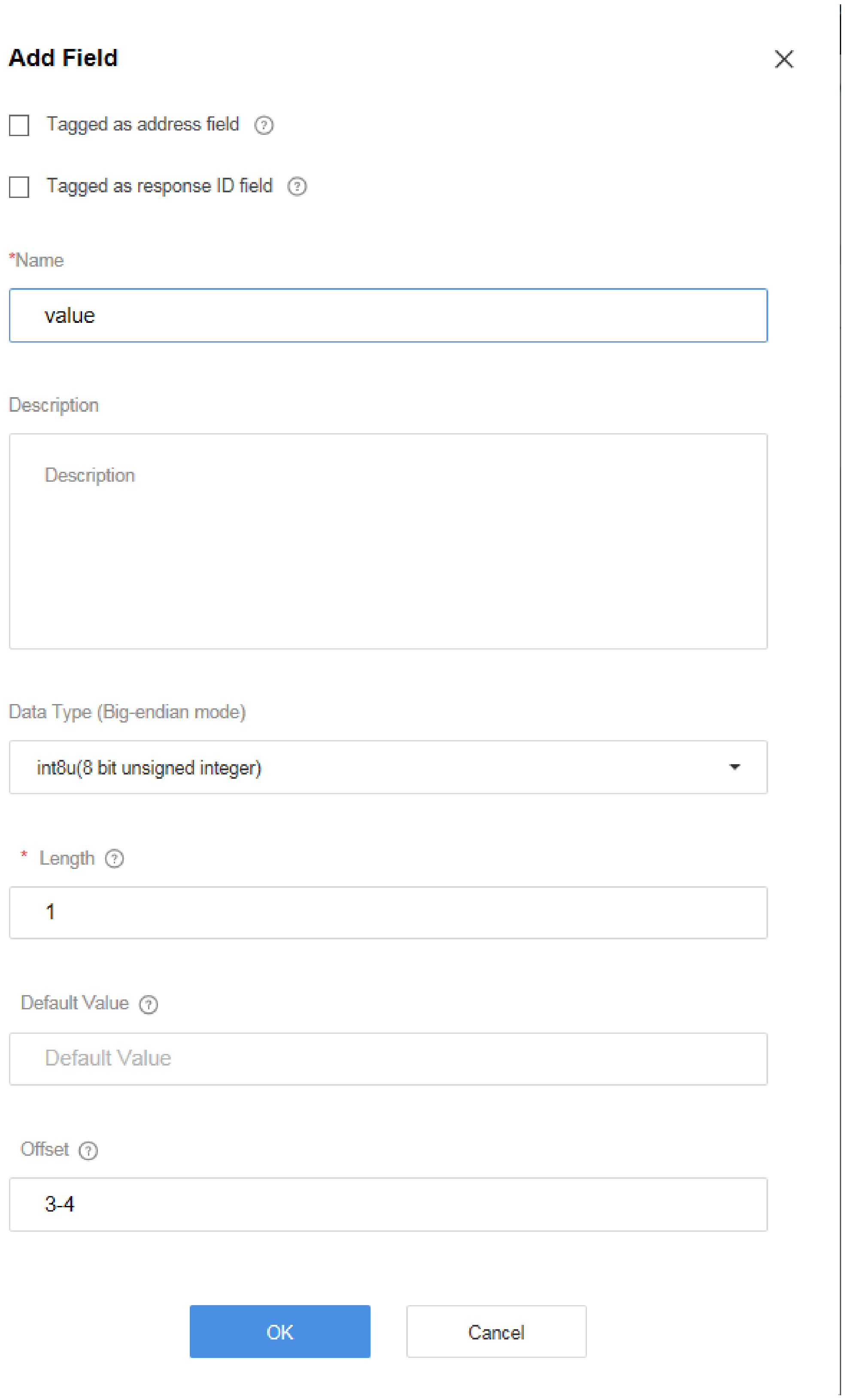
2 mid

Response Fields +

Step 15 In the **Add Field** dialog box, set the parameters and click **OK**.

- **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 profile file.



Add Field

☐ Tagged as address field

☐ Tagged as response ID field

*Name: value

Description: Description

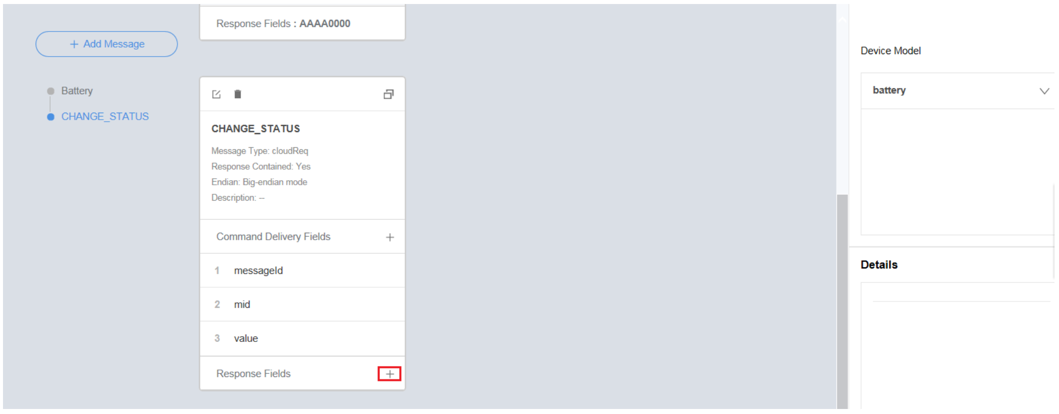
Data Type (Big-endian mode): int8u(8 bit unsigned integer)

* Length: 1

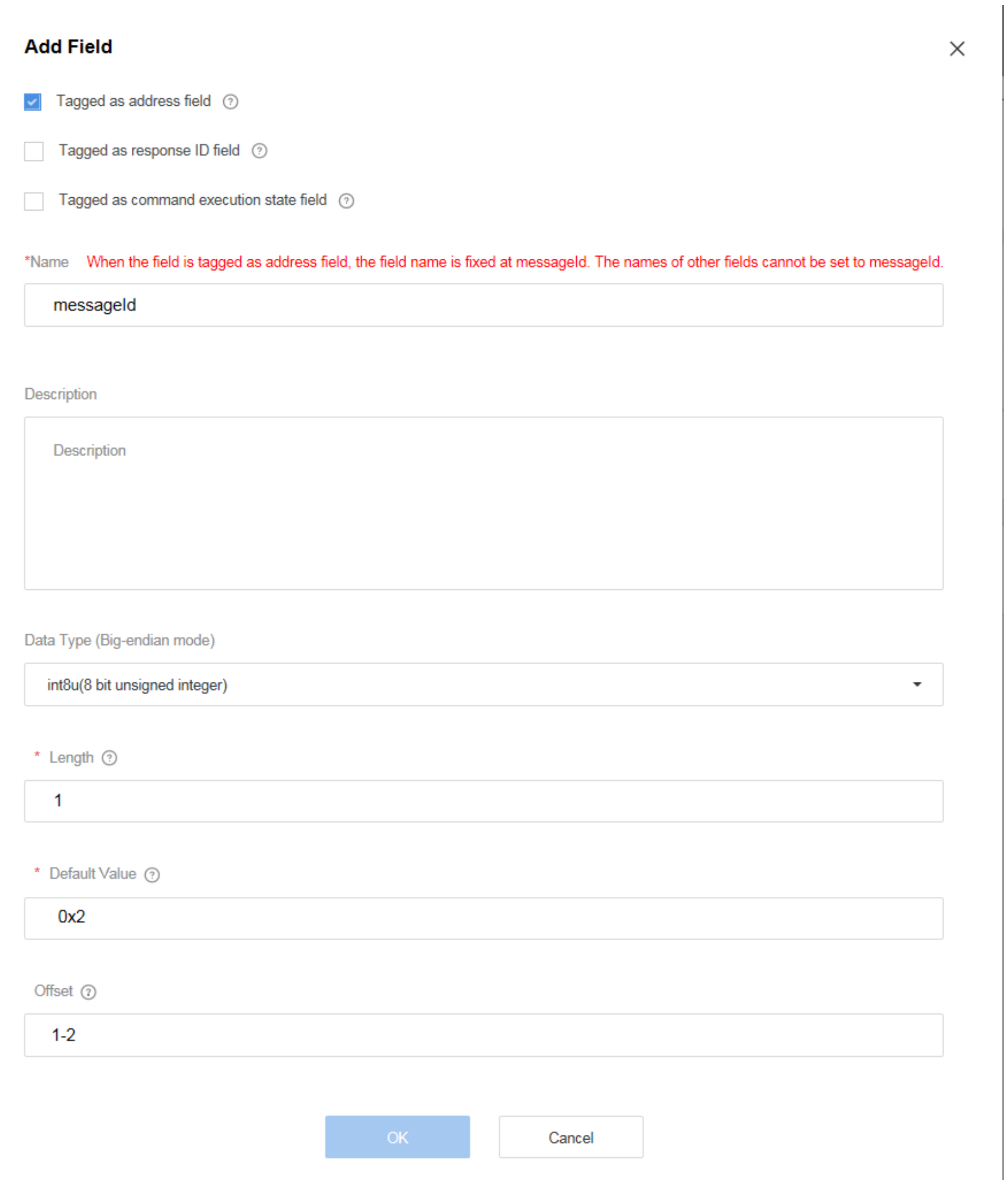
Default Value: Default Value

Offset: 3-4

Step 16 Click + next to **Response Fields**.



Step 17 In the **Add Field** dialog box, select **Tagged as address field**. Other parameters are set automatically. Click **OK**.


Add Field

[✓] Tagged as address field

[ ] Tagged as response ID field

[ ] Tagged as command execution state field

*Name When the field is tagged as address field, the field name is fixed at messageId. The names of other fields cannot be set to messageId.

messageId

Description

Description

Data Type (Big-endian mode)

int8u(8 bit unsigned integer)

* Length

1

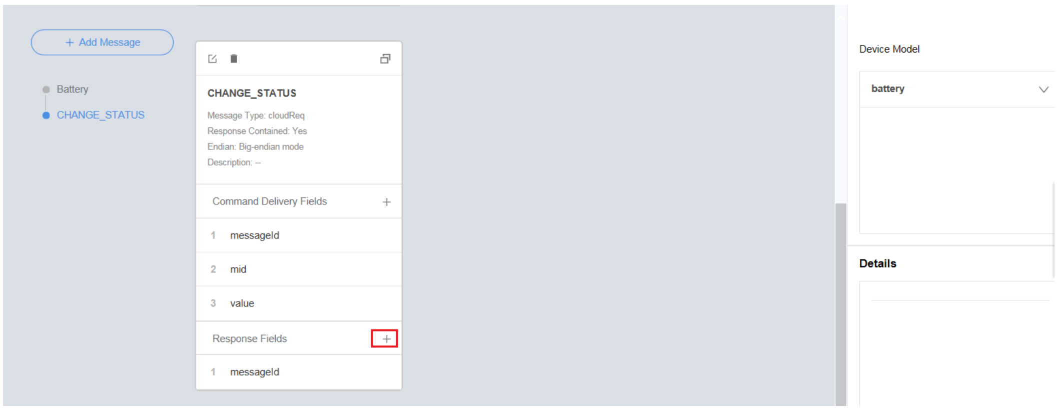
* Default Value

0x2

Offset

1-2

Step 18 Click + next to **Response Fields**.



CHANGE_STATUS
Message Type: cloudReq
Response Contained: Yes
Endian: Big-endian mode
Description: --

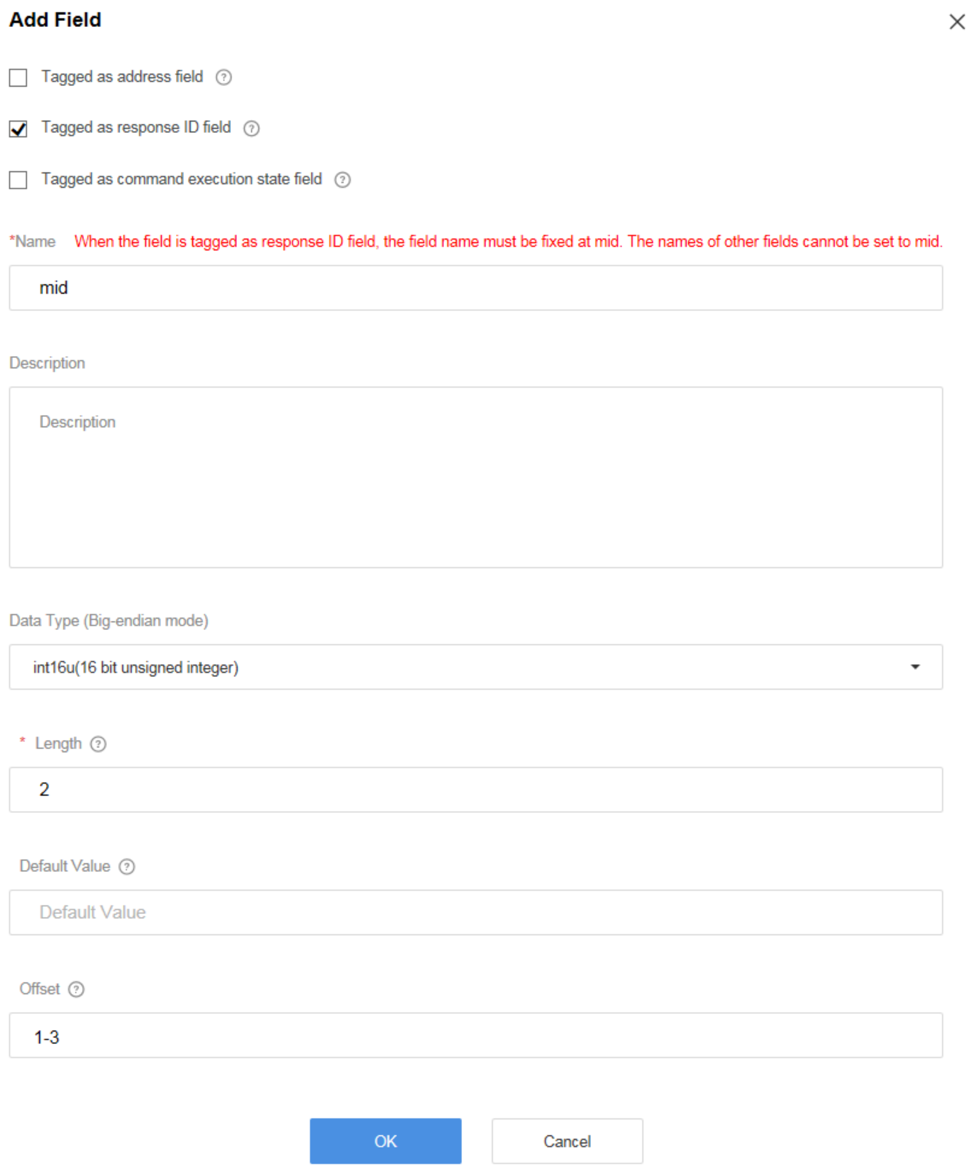
Command Delivery Fields

1 messageId
2 mid
3 value

Response Fields

1 messageId

Step 19 In the **Add Field** dialog box, select **Tagged as response ID field**. Other parameters are set automatically. Click **OK**.



Add Field

- [ ] Tagged as address field
- [x] Tagged as response ID field
- [ ] Tagged as command execution state field

*Name When the field is tagged as response ID field, the field name must be fixed at mid. The names of other fields cannot be set to mid.

mid

Description

Description

Data Type (Big-endian mode)

int16u(16 bit unsigned integer)

* Length

2

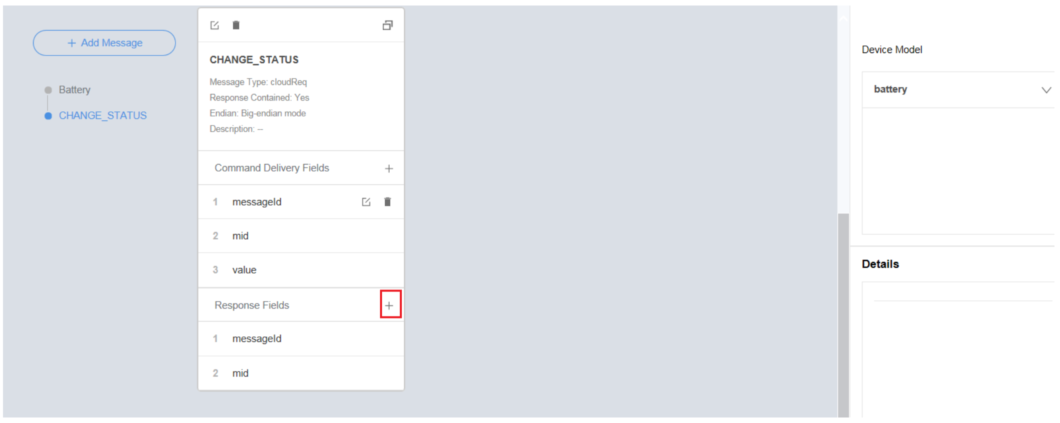
Default Value

Default Value

Offset

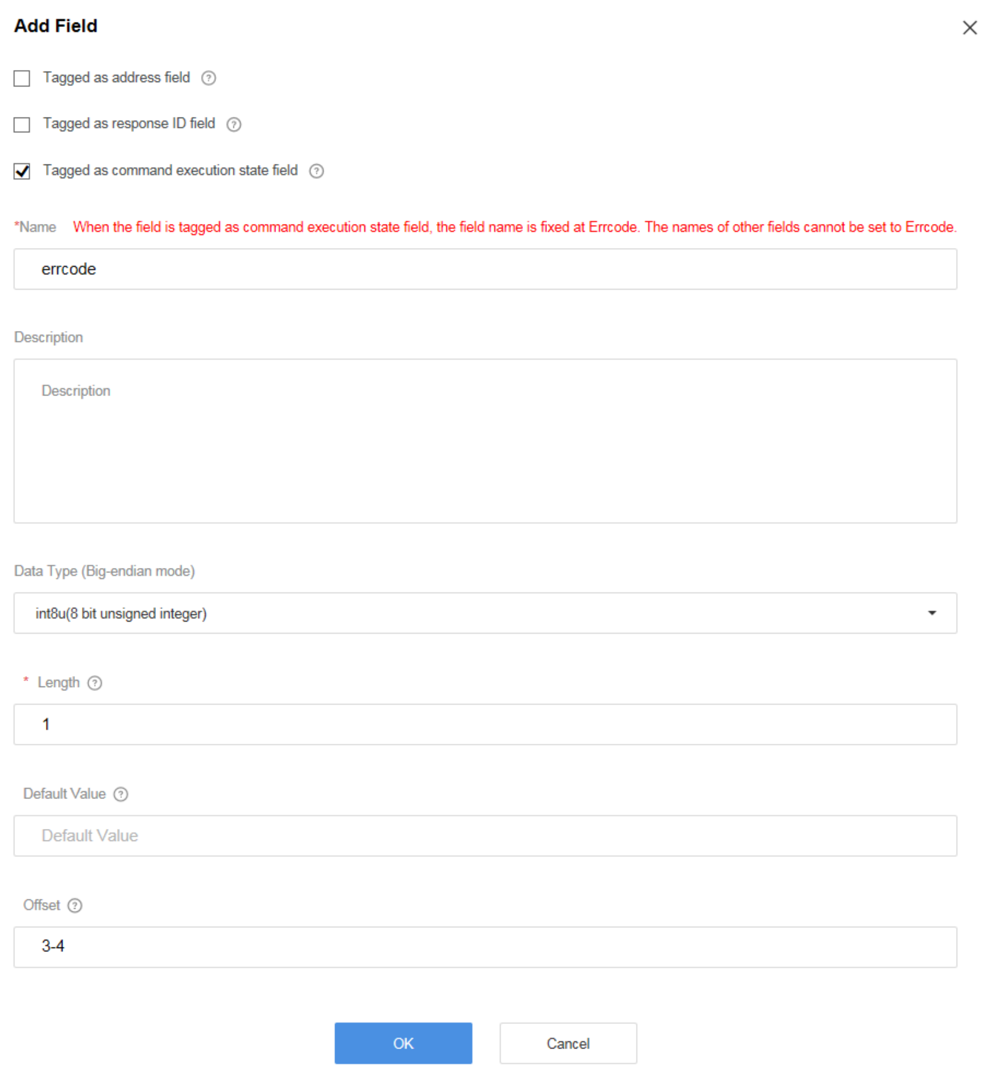
1-3

Step 20 Click + next to **Response Fields**.



Step 21 In the **Add Field** dialog box, select **Tagged as command execution state field**, set the other parameters, and click **OK**.

- The value of **Name** is automatically populated.
- **Data Type** is configured based on the actual command response and must match the type of the corresponding field defined in the profile file.



Add Field

- [ ] Tagged as address field
- [ ] Tagged as response ID field
- [x] Tagged as command execution state field

*Name When the field is tagged as command execution state field, the field name is fixed at Errcode. The names of other fields cannot be set to Errcode.

errcode

Description: Description

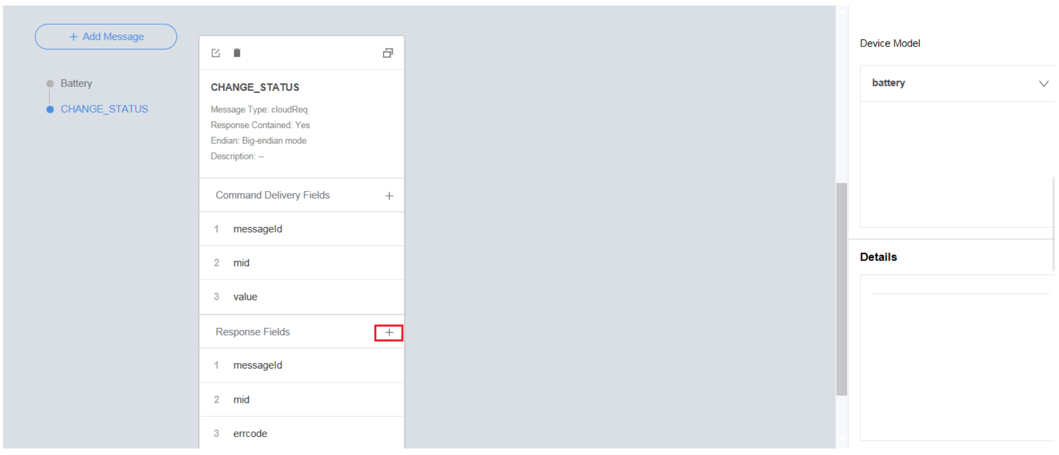
Data Type (Big-endian mode): int8u(8 bit unsigned integer)

* Length: 1

Default Value: Default Value

Offset: 3-4

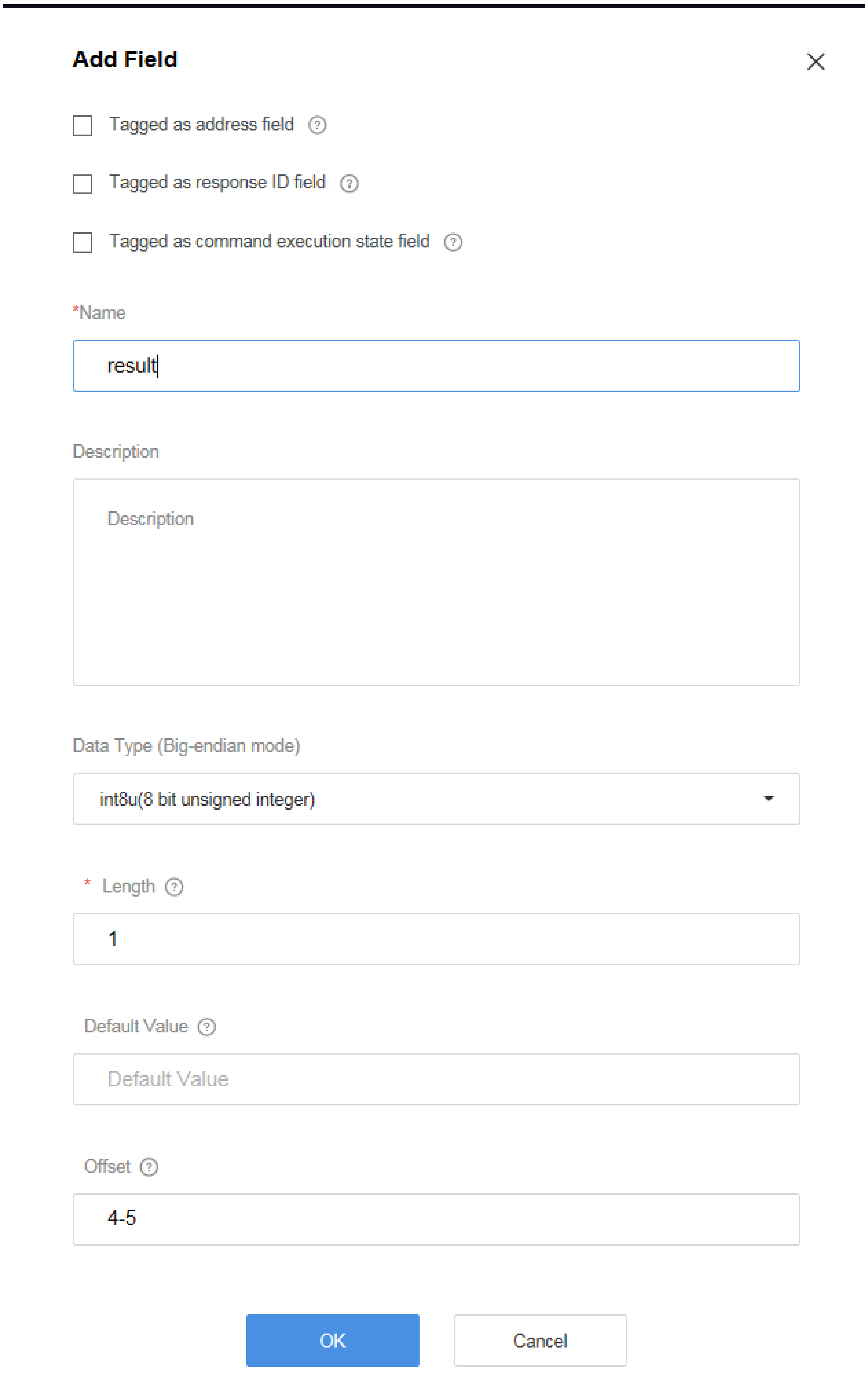
Step 22 Click + next to **Response Fields**.



Step 23 In the **Add Field** dialog box, set the parameters and click **OK**.

- **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 profile file.



☐ Tagged as address field

☐ Tagged as response ID field

☐ Tagged as command execution state field

*Name: result

Description: Description

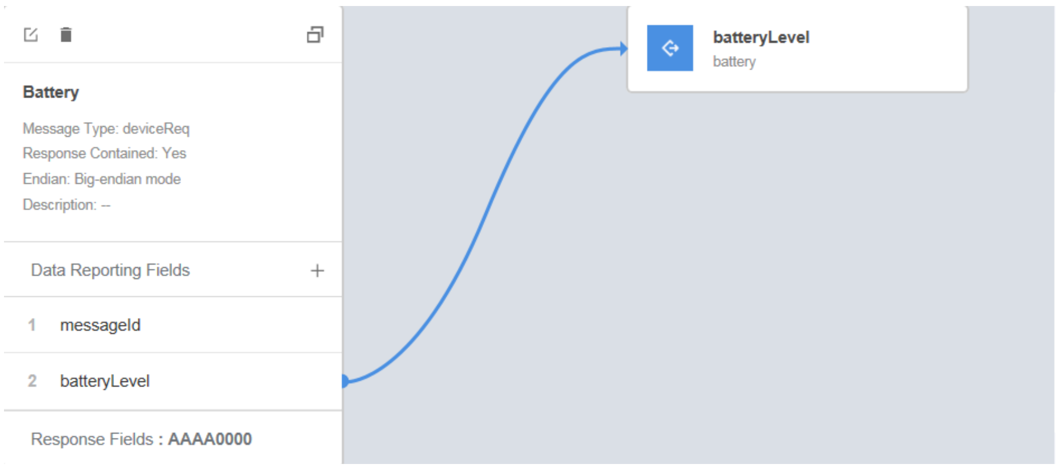
Data Type (Big-endian mode): int8u(8 bit unsigned integer)

* Length: 1

Default Value: Default Value

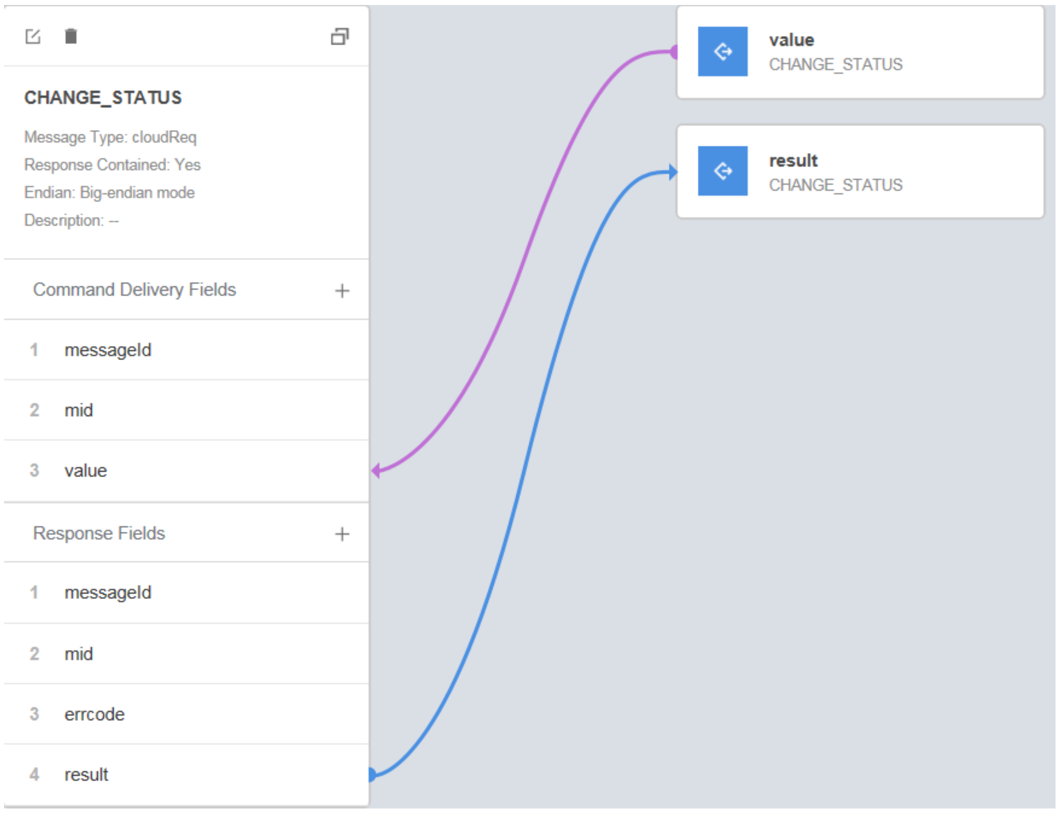
Offset: 4-5

Step 24 Map the property fields, command fields, and response fields in **Device Model** on the right with the fields in the data reporting message, command delivery message, and command response.



Battery

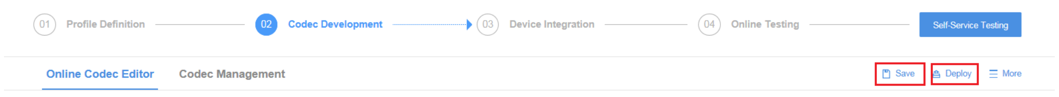
Message Type: deviceReq
Response Contained: Yes
Endian: Big-endian mode
Description: --



CHANGE_STATUS

Message Type: cloudReq
Response Contained: Yes
Endian: Big-endian mode
Description: --

Step 25 Click **Save** and then **Deploy** to deploy the codec on the IoT platform.



----End

1.4.2 Offline Development

1.4.2.1 Preparing the Development Environment

Downloading Eclipse

Download the Eclipse installation package and decompress it to a local directory. You can use the software without installation.

Eclipse is available on the official website at **http://www.eclipse.org/downloads**.

Downloading the Maven Plug-in

Download the Maven plug-in package (in .zip format) and decompress it to a local directory.

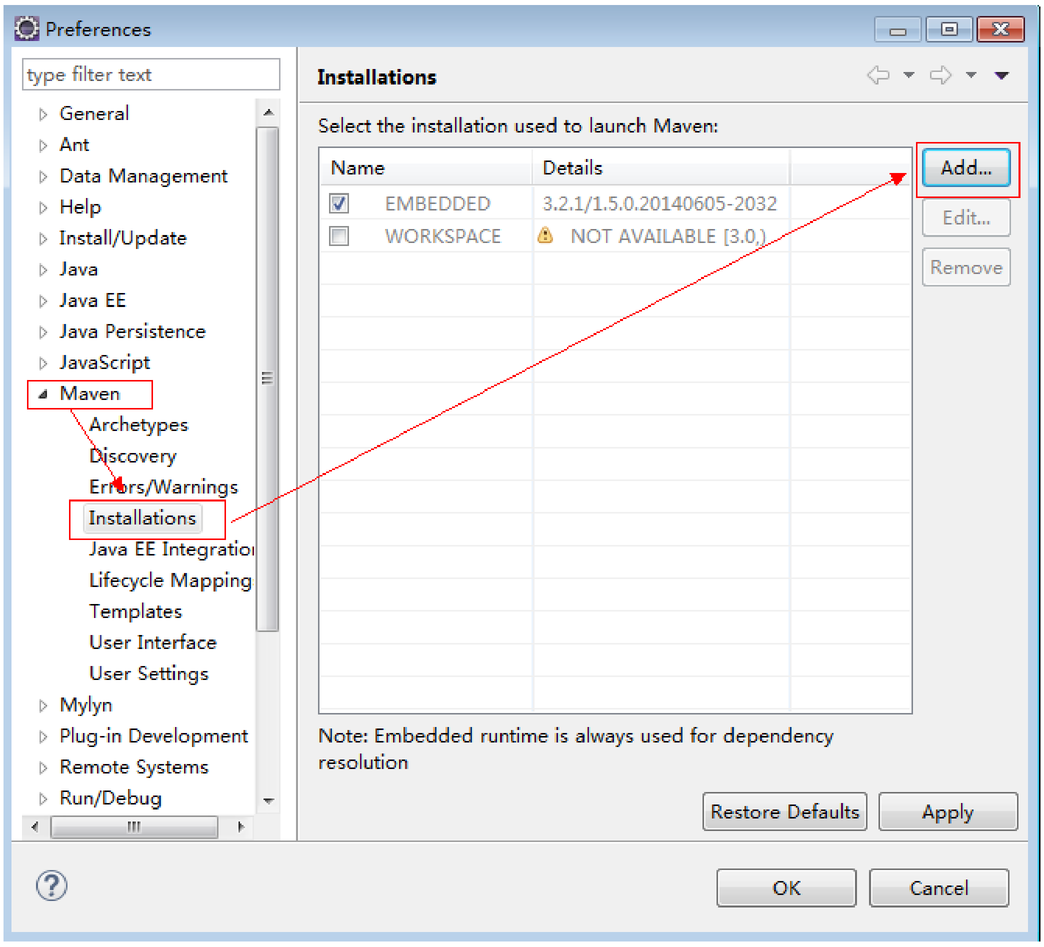
Maven is available on the official website at **http://maven.apache.org/download.cgi**.

Configuring the Maven Plug-in

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**.

Figure 1-3 Configuring Maven plug-in 1



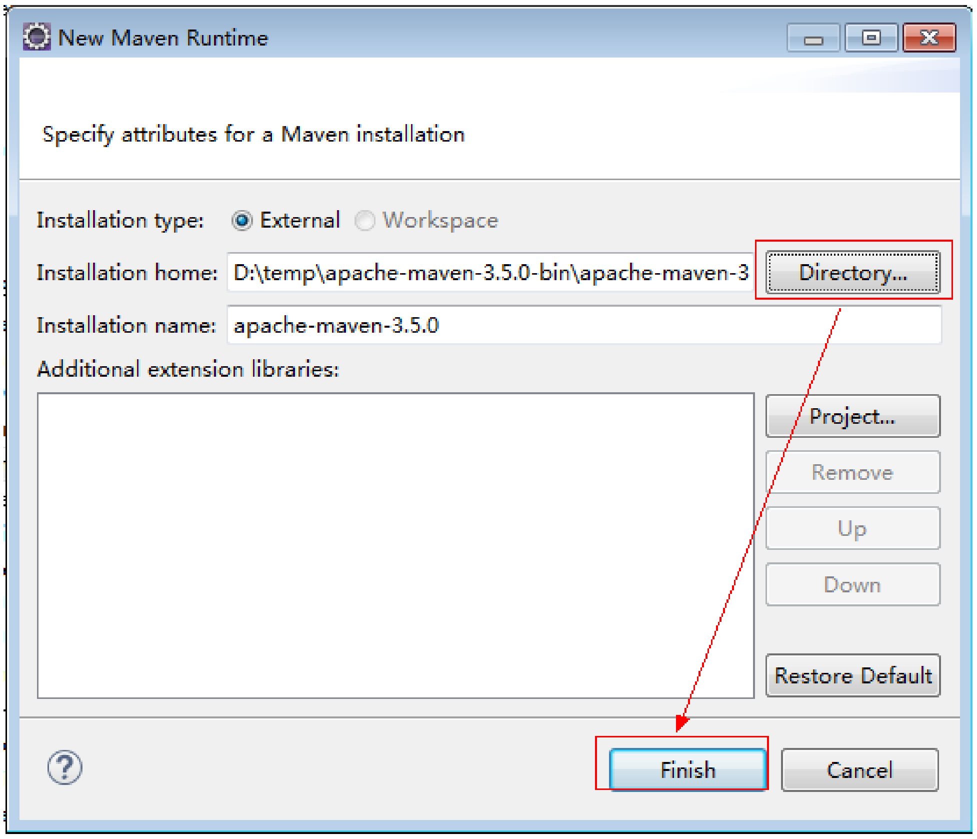
Select the installation used to launch Maven:

Name	Details
✓ EMBEDDED	3.2.1/1.5.0.20140605-2032
WORKSPACE	NOT AVAILABLE [3.0,)

Note: Embedded runtime is always used for dependency resolution

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

Figure 1-4 Configuring Maven plug-in 2



New Maven Runtime

Specify attributes for a Maven installation

Installation type: [✓ External] Workspace

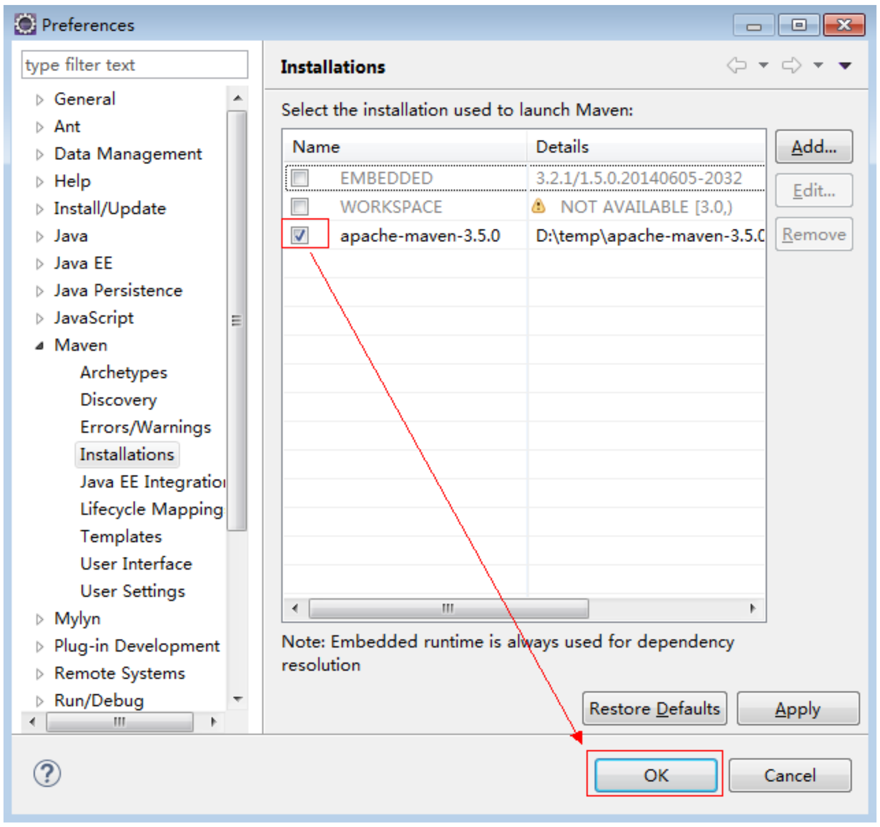
Installation home: D:\temp\apache-maven-3.5.0-bin\apache-maven-3

Installation name: apache-maven-3.5.0

Additional extension libraries:

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

Figure 1-5 Configuring Maven plug-in 3



Select the installation used to launch Maven:

	Name	Details
☐	EMBEDDED	3.2.1/1.5.0.20140605-2032
☐	WORKSPACE	NOT AVAILABLE [3.0,)
☑	apache-maven-3.5.0	D:\temp\apache-maven-3.5.0

Note: Embedded runtime is always used for dependency resolution

NOTE

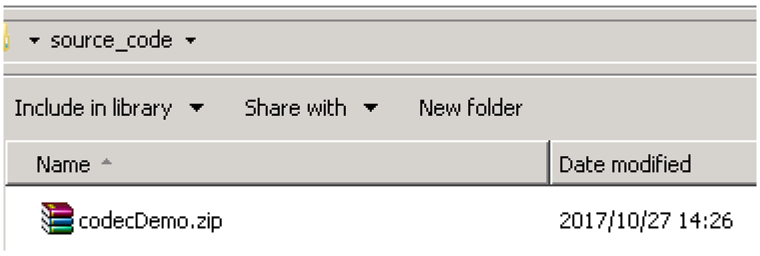
For details about how to install JDK and configure Java environment variables, see **Installing JDK 1.8** and **Configuring Java Environment Variables (Windows OS)**.

----End

1.4.2.2 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.

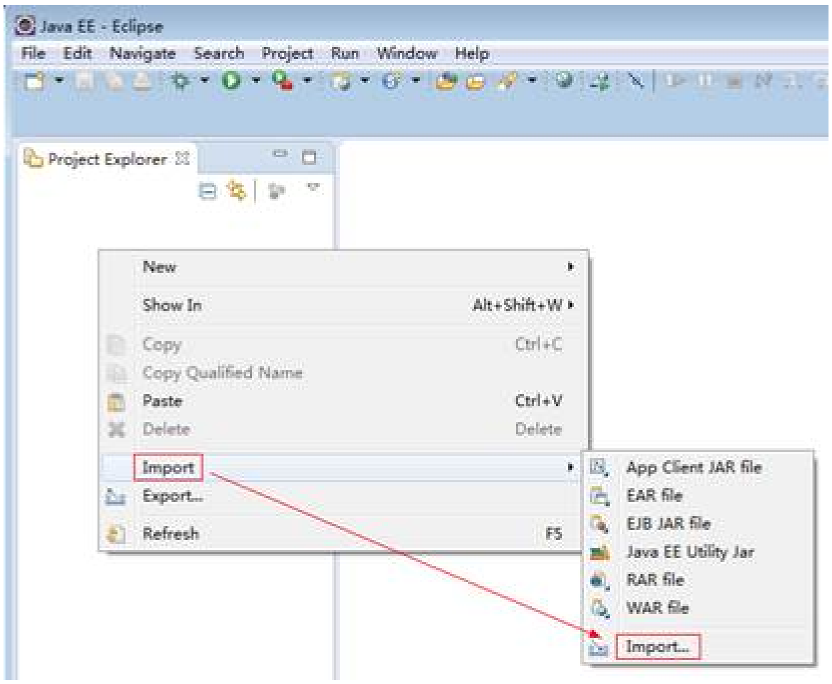
Figure 1-6 Position of the DEMO project of the codec



Name	Date modified
codecDemo.zip	2017/10/27 14:26

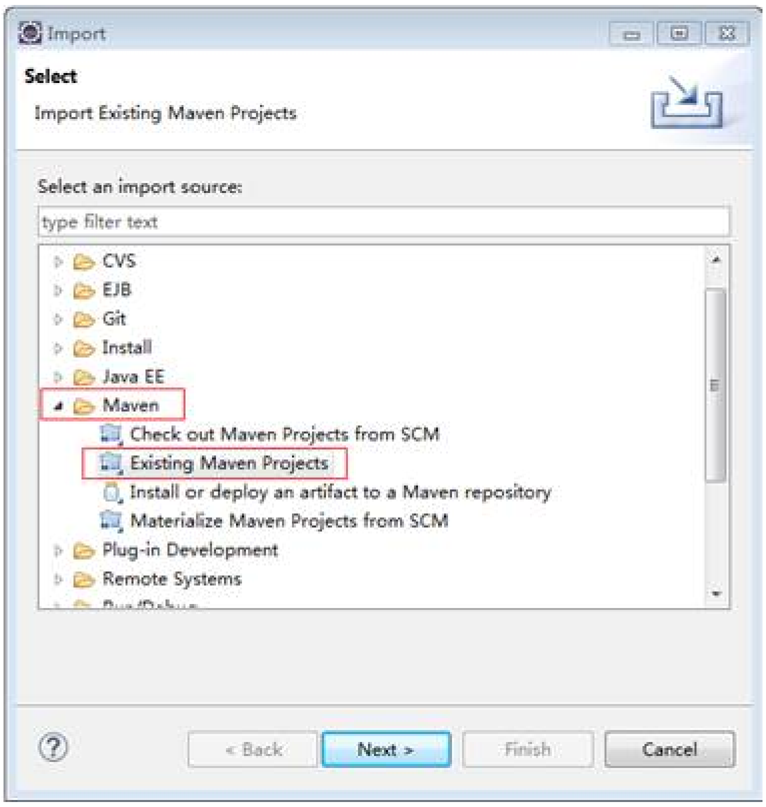
Step 2 Open Eclipse, right-click the blank area in **Project Explorer** on the left of Eclipse, and choose **Import** > **Import...**.

Figure 1-7 Importing DEMO project 1



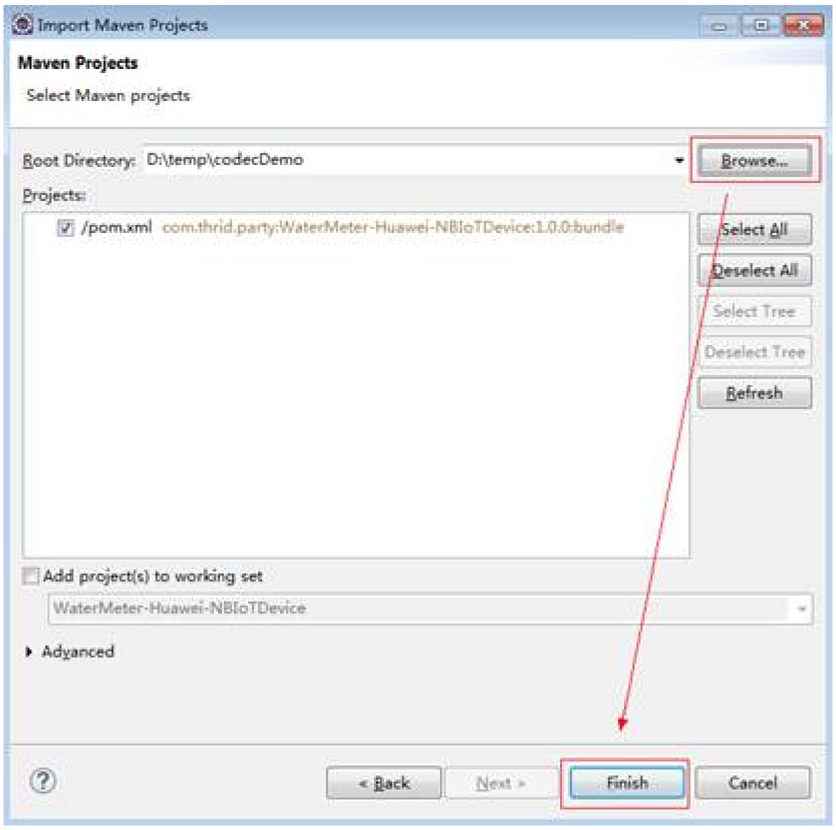
Step 3 Expand **Maven**, select **Existing Maven Projects**, and click **Next**.

Figure 1-8 Importing DEMO project 2



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

Figure 1-9 Importing DEMO project 3



----End

1.4.2.3 Developing a Codec

The Maven project architecture in the DEMO project does not need to be modified. To develop a codec, modify the DEMO project by following the instructions provided in **decode API Description**.

1.4.2.4 Packaging the Codec

This topic describes how to package the codec and prepare the package.

Packaging the Codec Using Maven

After the codec is programmed, use Maven to package the codec. On the Windows OS, perform the following steps:

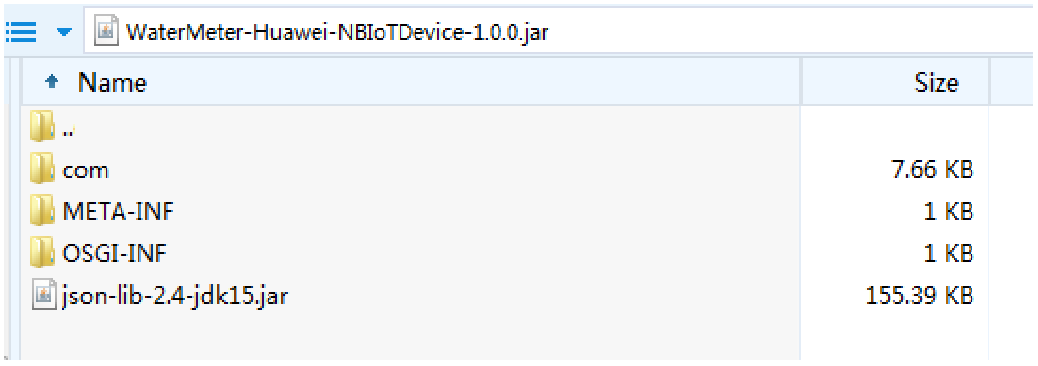
Step 1 Open the DOS window and access the directory where the **pom.xml** file is located.

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.

Figure 1-10 Structure of the .jar file



WaterMeter-Huawei-NBIoTDevice-1.0.0.jar

Name	Size
..	
com	7.66 KB
META-INF	1 KB
OSGI-INF	1 KB
json-lib-2.4-jdk15.jar	155.39 KB

- 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.

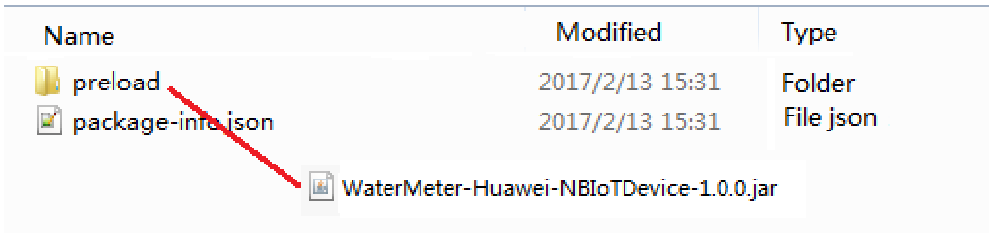
----End

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.

Figure 1-11 Structure of the codec package



Name	Modified	Type
preload	2017/2/13 15:31	Folder
package-info.json	2017/2/13 15:31	File json

WaterMeter-Huawei-NBIoTDevice-1.0.0.jar

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 1-2 Description of fields in the package-info.json file

Field	Description	Mandatory or Optional
specVersion	Specifies the version of the description file. The value is fixed at **1.0**.	Mandatory
fileName	Specifies the name of the software package. The value is fixed at **codec-demo**.	Mandatory
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**.	Mandatory
deviceType	Specifies the device type, which must be the same as that defined in the profile file.	Mandatory
manufacturerName	Specifies the manufacturer name, which must be the same as that defined in the profile file. Otherwise, the **package-info.json** file cannot be uploaded to the IoT platform.	Mandatory
model	Specifies the product model, which must be the same as that defined in the profile file.	Mandatory
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**.	Mandatory
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**.	Mandatory
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.	Optional
description	Specifies the self-defined description about the software package.	Optional
ignoreList	Specifies the list of bundles to be ignored. The default value is **null**.	Mandatory
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.	Mandatory

Table 1-3 Description of the bundles field

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

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

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

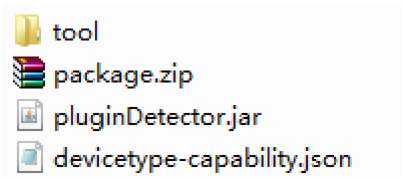
1.4.2.5 Inspecting the Quality of the Codec

After the codec is packaged, quality inspection is performed to check whether the codec is functioning properly.

Step 1 Obtain the **codec detection tool** from the IoT platform service provider.

Step 2 Save the **pluginDetector.jar** file, the **devicetype-capability.json** file in the profile file, and the **package.zip** and **tool** folders to be checked to the same directory.

Figure 1-12 Placing the files in the same directory



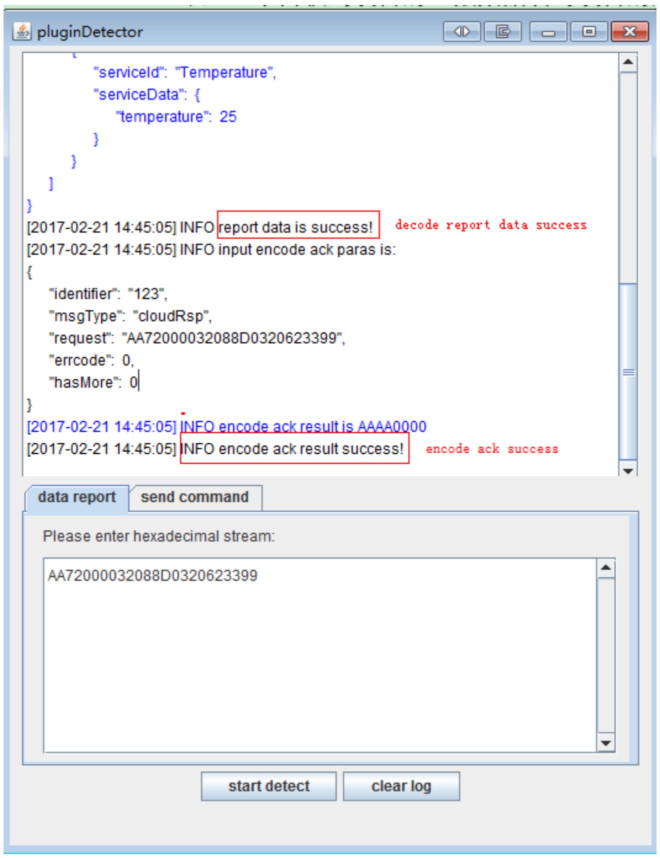
tool
package.zip
pluginDetector.jar
devicetype-capability.json

Step 3 Obtain a stream of reported device data, and enter the stream in hexadecimal format on the **data report** tab page of the detection tool, for example, AA72000032088D0320623399.

Step 4 Click **start detect** to view the decoded JSON data.

The log text box displays the decoded data. If **report data is success** is displayed, the decoding is successful. If **ERROR** is displayed, an error occurs during decoding.

Figure 1-13 Successful decoding of reported data

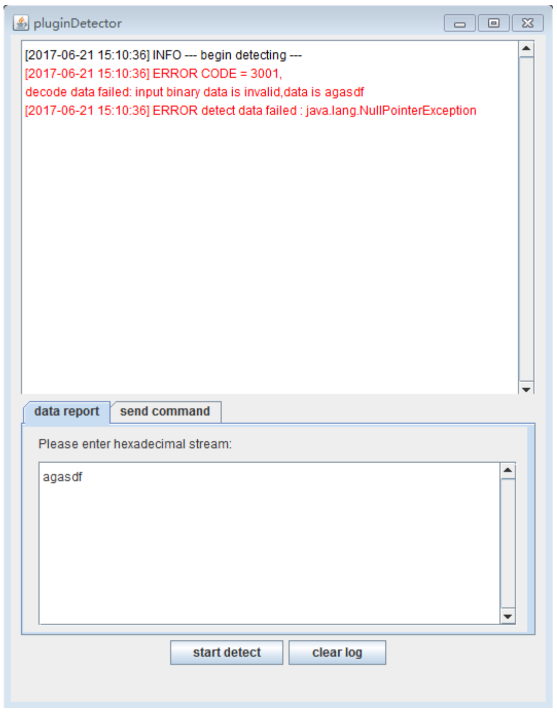


```
        "serviceId": "Temperature",
        "serviceData": {
            "temperature": 25
        }
    }
  ]
}
[2017-02-21 14:45:05] INFO report data is success!
[2017-02-21 14:45:05] INFO input encode ack paras is:
{
    "identifier": "123",
    "msgType": "cloudRsp",
    "request": "AA72000032088D0320623399",
    "errcode": 0,
    "hasMore": 0
}
[2017-02-21 14:45:05] INFO encode ack result is AAAA0000
[2017-02-21 14:45:05] INFO encode ack result success!
```

Please enter hexadecimal stream:

AA72000032088D0320623399

Figure 1-14 Failed decoding of reported data



```
[2017-06-21 15:10:36] INFO --- begin detecting ---
[2017-06-21 15:10:36] ERROR CODE = 3001,
decode data failed: input binary data is invalid,data is agasdf
[2017-06-21 15:10:36] ERROR detect data failed : java.lang.NullPointerException
```

```
agasdf
```

Step 5 After the decoding is successful, the detection tool continues to call the encode method of the codec package to encode a response.

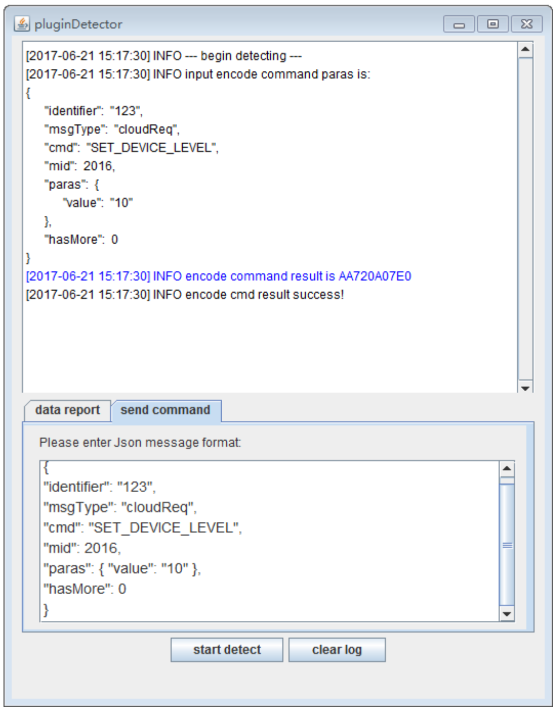
If **encode ack result success** is displayed, the response is encoded successfully.

Step 6 Obtain a command delivered by the application server. (The application server calls the API for creating device commands on the IoT platform to deliver the command.) Then, enter the command on the **data report** tab page of the detection tool.

Step 7 Click **start detect** of the detection tool. Then, the detection tool calls the encode API to encode a control command.

If **encode cmd result success** is displayed, the command is successfully encoded. If **ERROR** is displayed, an error occurs during the command encoding.

Figure 1-15 Successful encoding control command delivery



```
[2017-06-21 15:17:30] INFO --- begin detecting ---
[2017-06-21 15:17:30] INFO input encode command paras is:
{
    "identifier": "123",
    "msgType": "cloudReq",
    "cmd": "SET_DEVICE_LEVEL",
    "mid": 2016,
    "paras": {
        "value": "10"
    },
    "hasMore": 0
}
[2017-06-21 15:17:30] INFO encode command result is AA720A07E0
[2017-06-21 15:17:30] INFO encode cmd result success!
```

Please enter Json message format:

```
{
"identifier": "123",
"msgType": "cloudReq",
"cmd": "SET_DEVICE_LEVEL",
"mid": 2016,
"paras": { "value": "10" },
"hasMore": 0
}
```

Order example:

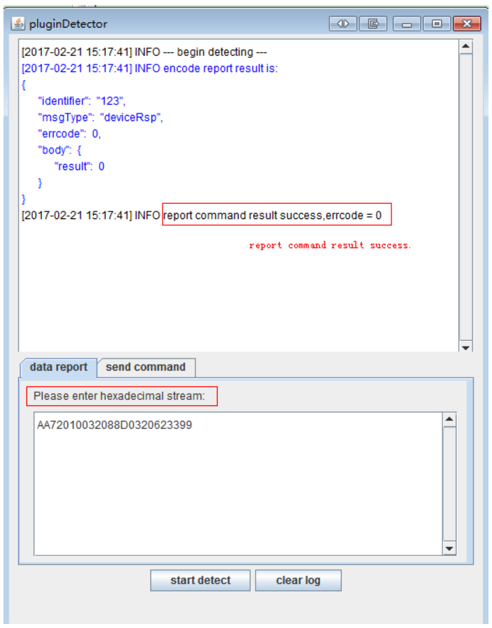
```
{
    "identifier": "123",
    "msgType": "cloudReq",
    "serviceId": "NBWaterMeterCommon",
    "cmd": "SET_DEVICE_LEVEL",
    "mid": 2016,
    "paras": {
        "value": "10"
    },
    "hasMore": 0
}
```

Step 8 Obtain a stream of reported device command execution results, and enter the stream in hexadecimal format on the **data report** tab page of the detection tool, for example, AA7201000107E0.

Step 9 Click **start detect** to view the decoded JSON data.

The log text box displays the decoded data. If **report command result success** is displayed, the decoding is successful. If **ERROR** is displayed, an error occurs during decoding.

Figure 1-16 Successful decoding of the command execution result



```
[2017-02-21 15:17:41] INFO --- begin detecting ---
[2017-02-21 15:17:41] INFO encode report result is:
{
    "identifier":  "123",
    "msgType":  "deviceRsp",
    "errcode":  0,
    "body":  {
        "result":  0
    }
}
[2017-02-21 15:17:41] INFO report command result success,errcode = 0
```

----End

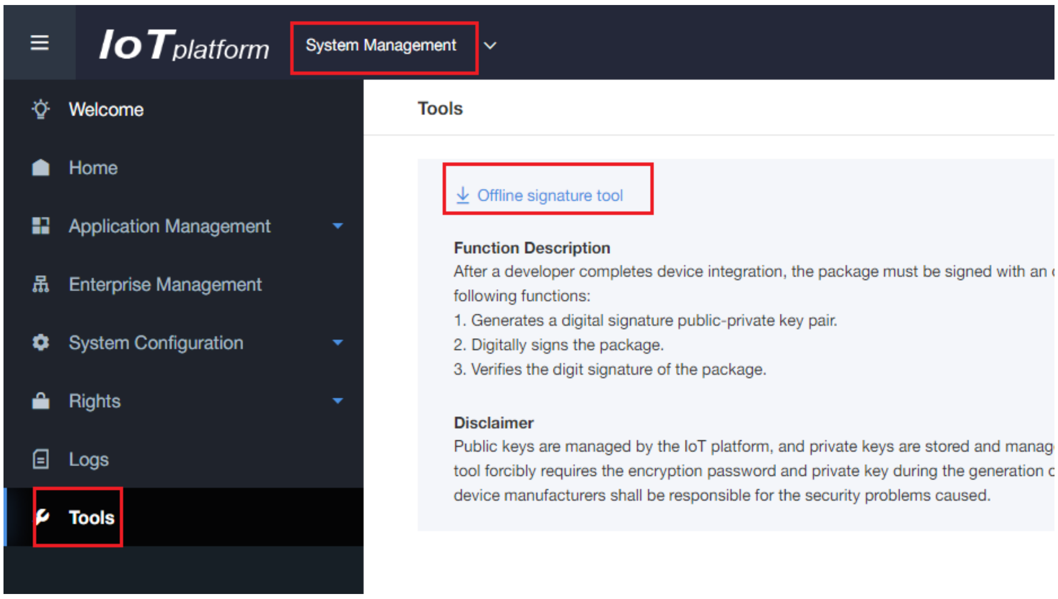
1.4.2.6 Signing the Codec Package with an Offline Signature

After the codec is developed, sign the codec package before installing it on the IoT platform. To sign the package, download Huawei Offline Signtool.

Step 1 Log in to the Management Portal.

Step 2 Choose **System Management** > **Tools**, and click **Offline signature tool** to obtain the tool.

Figure 1-17 Downloading the offline signature tool

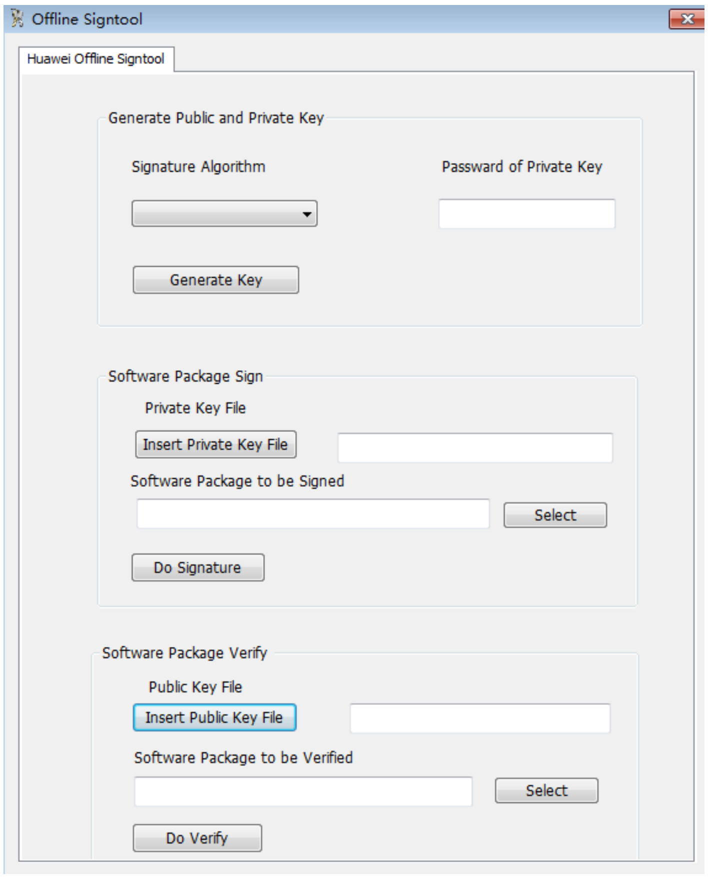


Function Description
After a developer completes device integration, the package must be signed with an
following functions:
1. Generates a digital signature public-private key pair.
2. Digitally signs the package.
3. Verifies the digit signature of the package.

Disclaimer
Public keys are managed by the IoT platform, and private keys are stored and manag
tool forcibly requires the encryption password and private key during the generation
device manufacturers shall be responsible for the security problems caused.

Step 3 Decompress the **signtool.zip** file and double-click **signtool.exe** to run Huawei Offline Signtool.

Figure 1-18 Running Huawei Offline Signtool



Step 4 In the **Generate Public and Private Key** area, select a value for **Signature algorithm**, set **Password of Private key**, and click **Generate Key**. In the dialog box displayed, select the directory to save the key files and click **OK**.

Set **Signature Algorithm** as required. Currently, two signature algorithms are available:

- ECDSA_256K1+SHA256
- RSA2048+SHA256

When setting **Password of Private Key**, ensure that the password complexity meets the following conditions:

- The password must contain at least six characters.
- The password must contain at least two types of the following characters:
  - A-Z
  - a-z
  - 0-9
  - :~`@#$%^&*()-_=+|?/<>[]{},.;'!"

The public and private key files are generated in the storage directory.

- Public key file: public.pem
- Private key file: private.pem

Step 5 In the **Software Package Sign** area, import the private key file, enter the password, and click **OK**. The password is the value of **Password of Private Key** set in Step 4.

Step 6 Select the software package to be signed and click **Do Signature**.

If the digital signature is successful, the software package named **xxx_signed.xxx** with a digital signature is generated in the directory where the original software package is located.

NOTE

The offline signature tool can sign only the packages in .zip format with a digital signature.

Step 7 In the **Software Package Verify** area, import the public key file and click **OK**.

Step 8 Select the software package (generated in **Step 6**) that requires signature verification and click **Do Verify**.

- If **Verify Success!** is displayed, the signature verification is successful.
- If **Verify Error!** is displayed, the signature verification fails.

  NOTE

  During software package verification, the path for storing the signed software package must not contain Chinese characters.

----End

1.4.3 Codec Development Examples

1.4.3.1 Codec for Data Reporting and Command Delivery

Scenarios

A smoke detector provides the following functions:

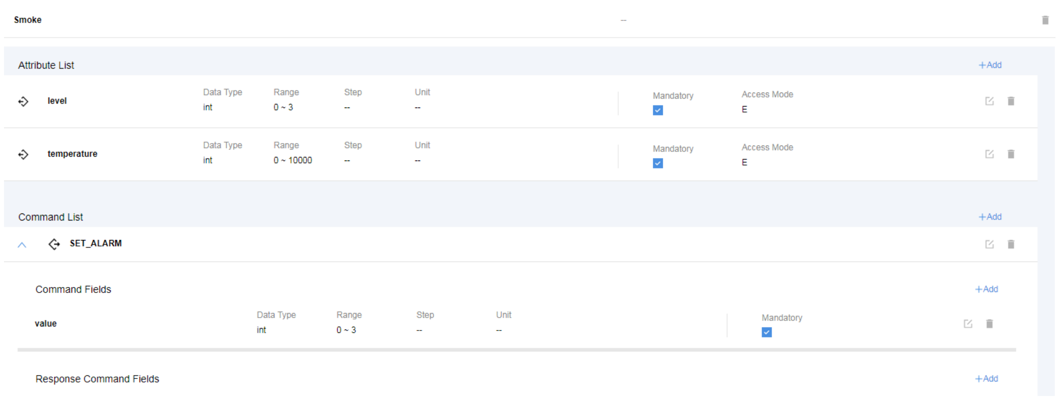
- Reporting smoke alarms (fire severity) and temperature
- Remote command, which can enable the alarm function remotely

  For example, the smoke detector can report the temperature on the fire scene and remotely trigger the smoke alarm for evacuation.

Defining the Profile File

Define the profile file in the development space of the smoke sensor.

- 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.


Smoke --

Attribute List +Add

	Data Type	Range	Step	Unit	Mandatory	Access Mode
level	int	0 ~ 3	--	--	✓	E
temperature	int	0 ~ 10000	--	--	✓	E

Command List +Add

SET_ALARM

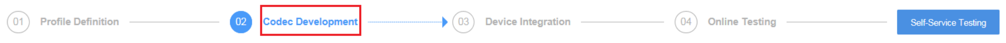
Command Fields +Add

	Data Type	Range	Step	Unit	Mandatory
value	int	0 ~ 3	--	--	✓

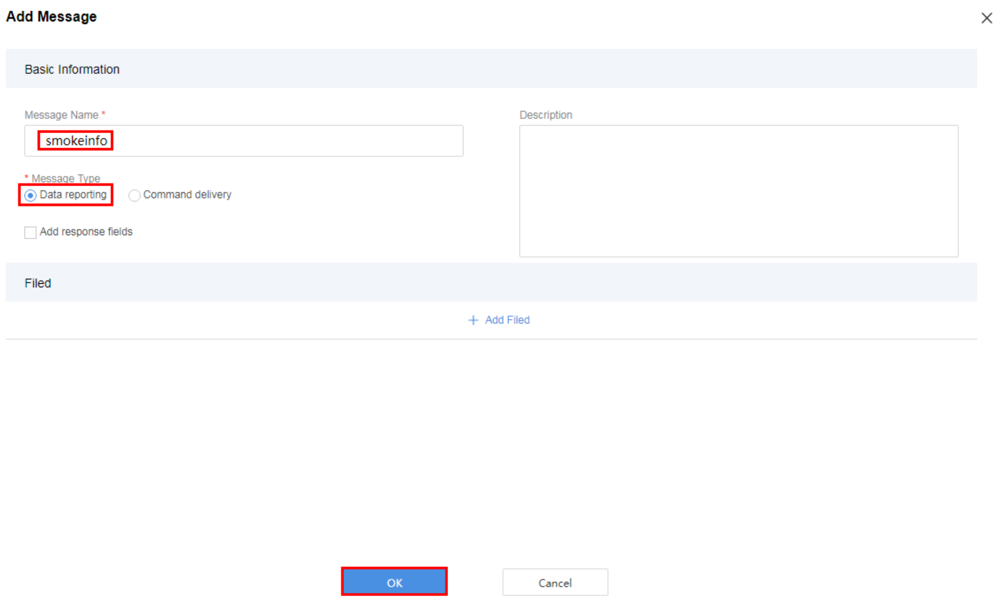
Response Command Fields +Add

Developing a Codec

Step 1 In the development space of the smoke sensor, click **Codec Development**.



Step 2 Configure a data reporting message.



Add Message

Basic Information

Message Name * smokeinfo

Description

* Message Type [✓ Data reporting] Command delivery

Add response fields

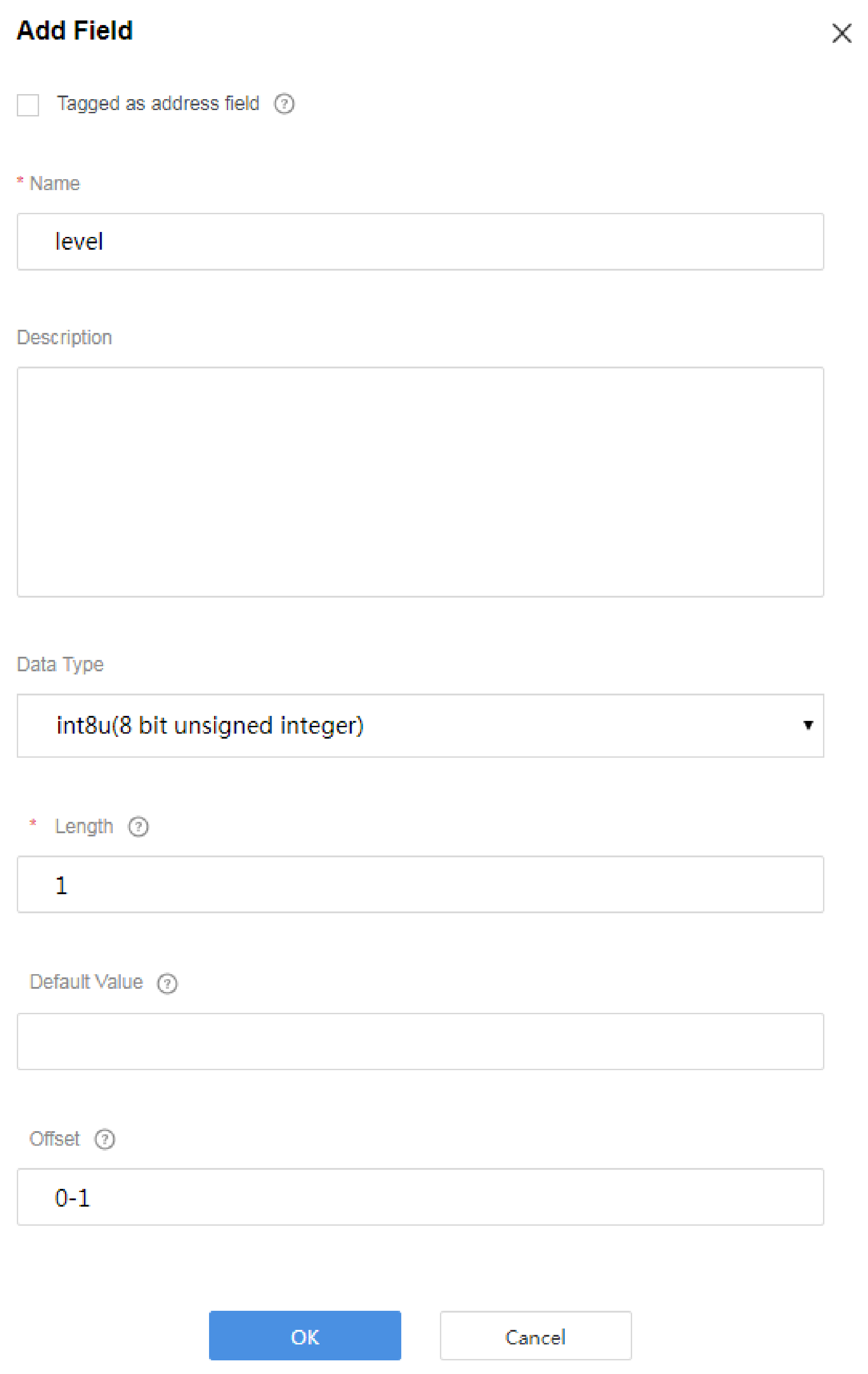
Filed

+ Add Filed

OK Cancel

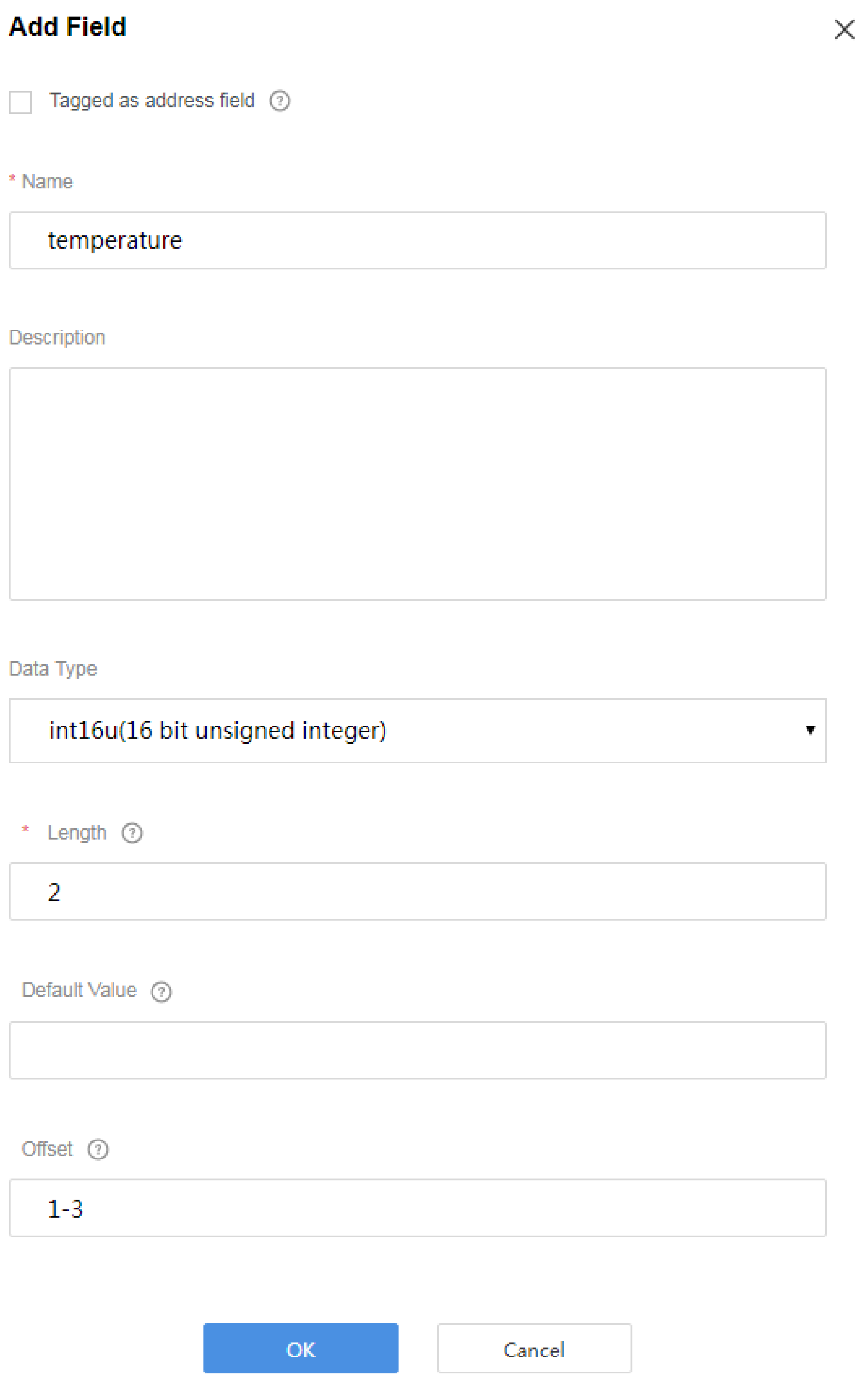
Add a **level** field to indicate the fire severity.

- **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 profile file.
- The values of **Length** and **Offset** are automatically filled based on **Data Type**.



* Name	level
Description	
Data Type	int8u(8 bit unsigned integer)
* Length	1
Default Value	
Offset	0-1

Add the **temperature** field to indicate the temperature at the fire scene. In the profile file, the maximum value of **temperature** is **1000**. Therefore, set the data type of the **temperature** field to **int16u** in the codec to meet the value range requirement of **temperature**.


Add Field

Tagged as address field

* Name

temperature

Description

Data Type

int16u(16 bit unsigned integer)

* Length

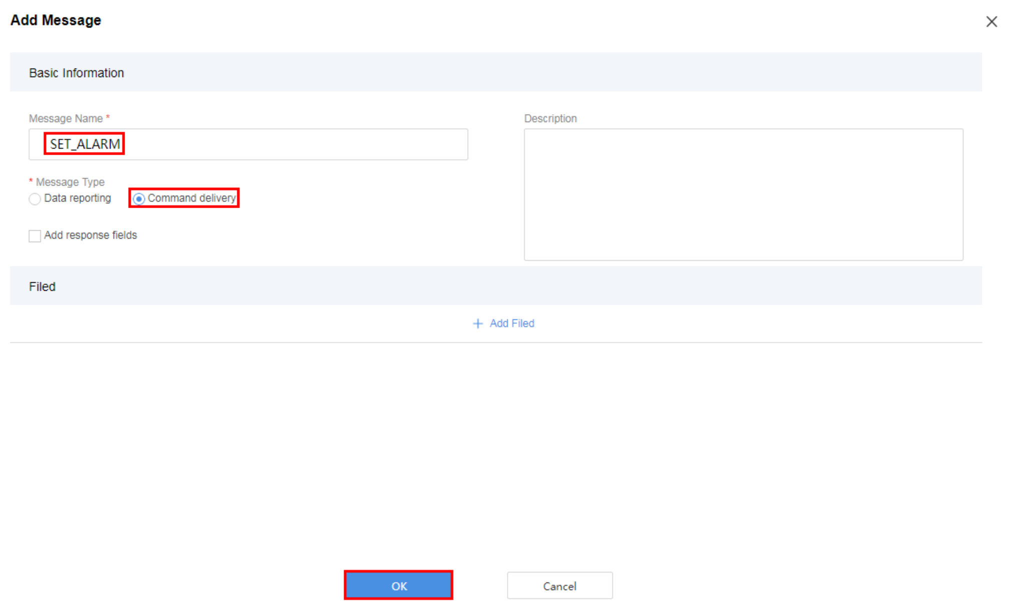
2

Default Value

Offset

1-3

Step 3 Configure a command delivery message.


Add Message

Basic Information

Message Name *: SET_ALARM

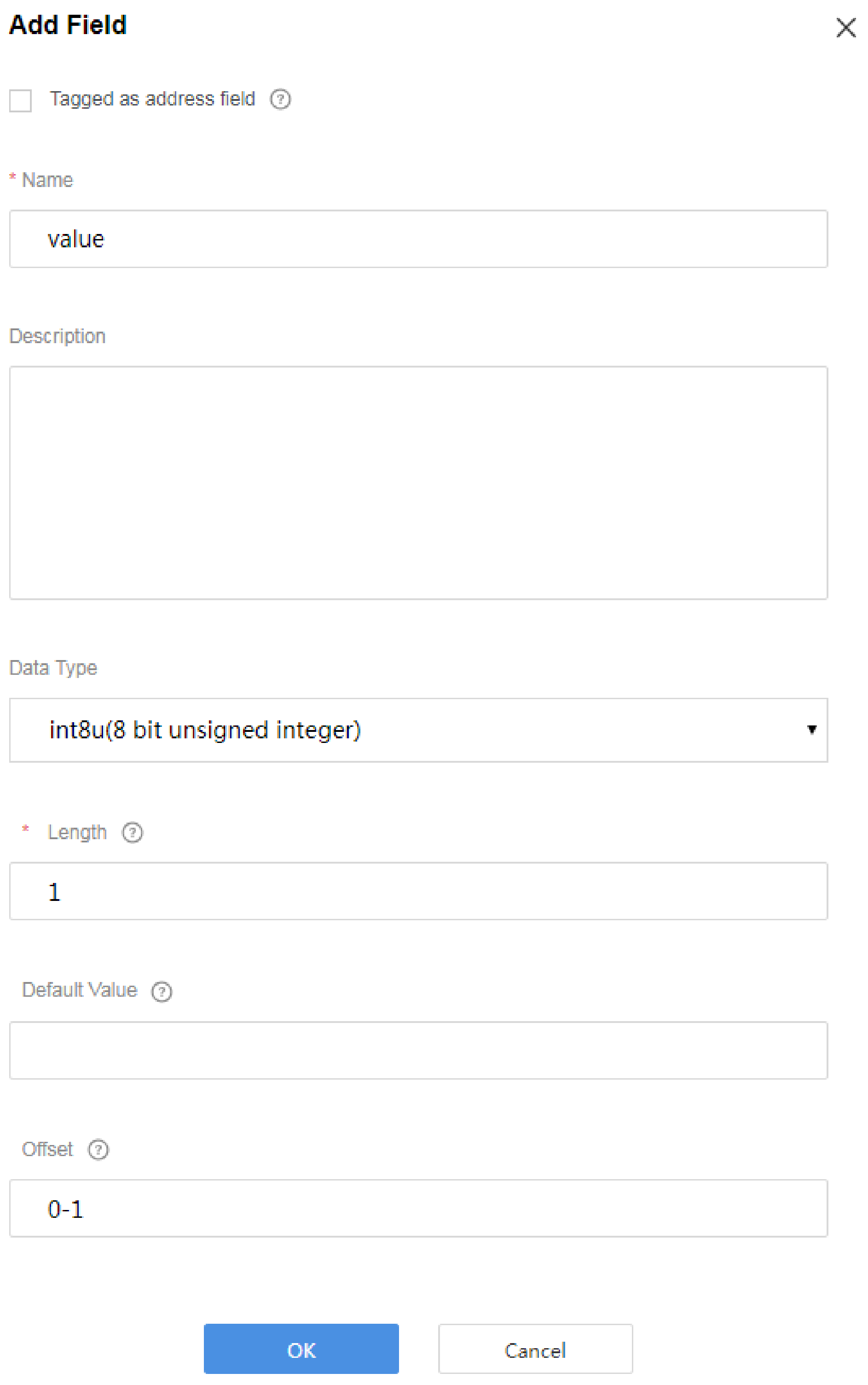
Description:

* Message Type: Data reporting [✓ Command delivery]

Add response fields

Filed

Add the **value** field to indicate the parameter value of the delivered command.



Add Field

☐ Tagged as address field

* Name: value

Description:

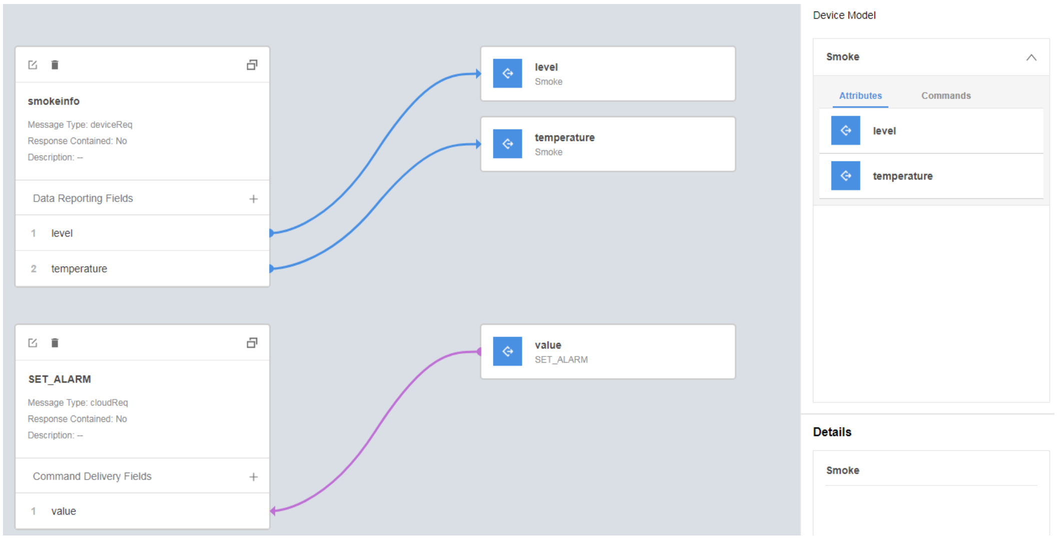
Data Type: int8u(8 bit unsigned integer)

* Length: 1

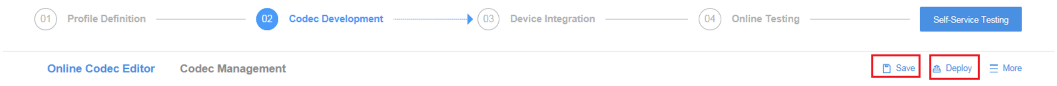
Default Value:

Offset: 0-1

Step 4 Map the property fields and command fields in **Device Model** on the right with the fields in the data reporting message and command delivery message.



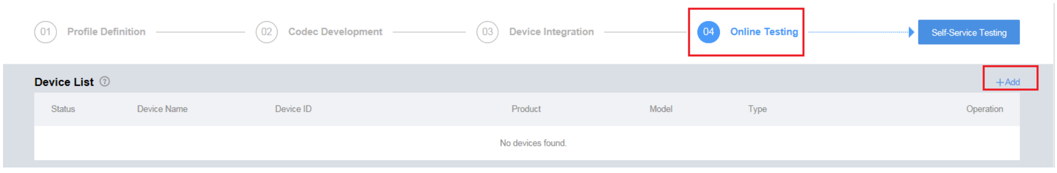
Step 5 Click **Save** and then **Deploy** to deploy the codec on the IoT platform.



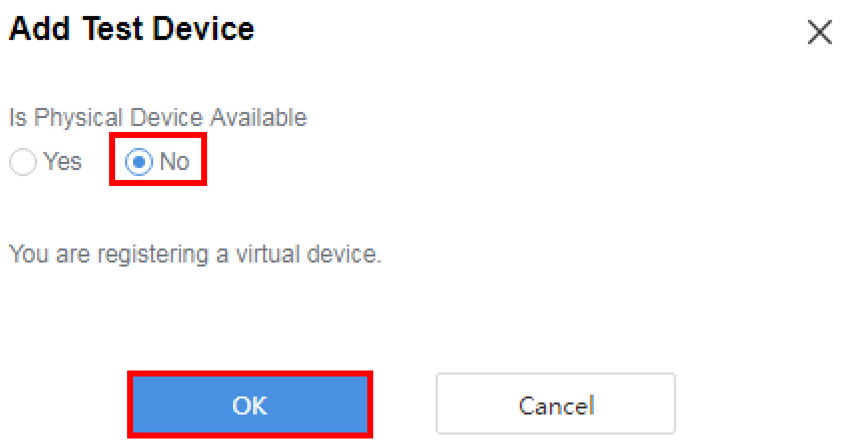
----End

Testing the Codec

Step 1 In the development space of the smoke sensor, click **Online Testing** and add a virtual device to test the codec.

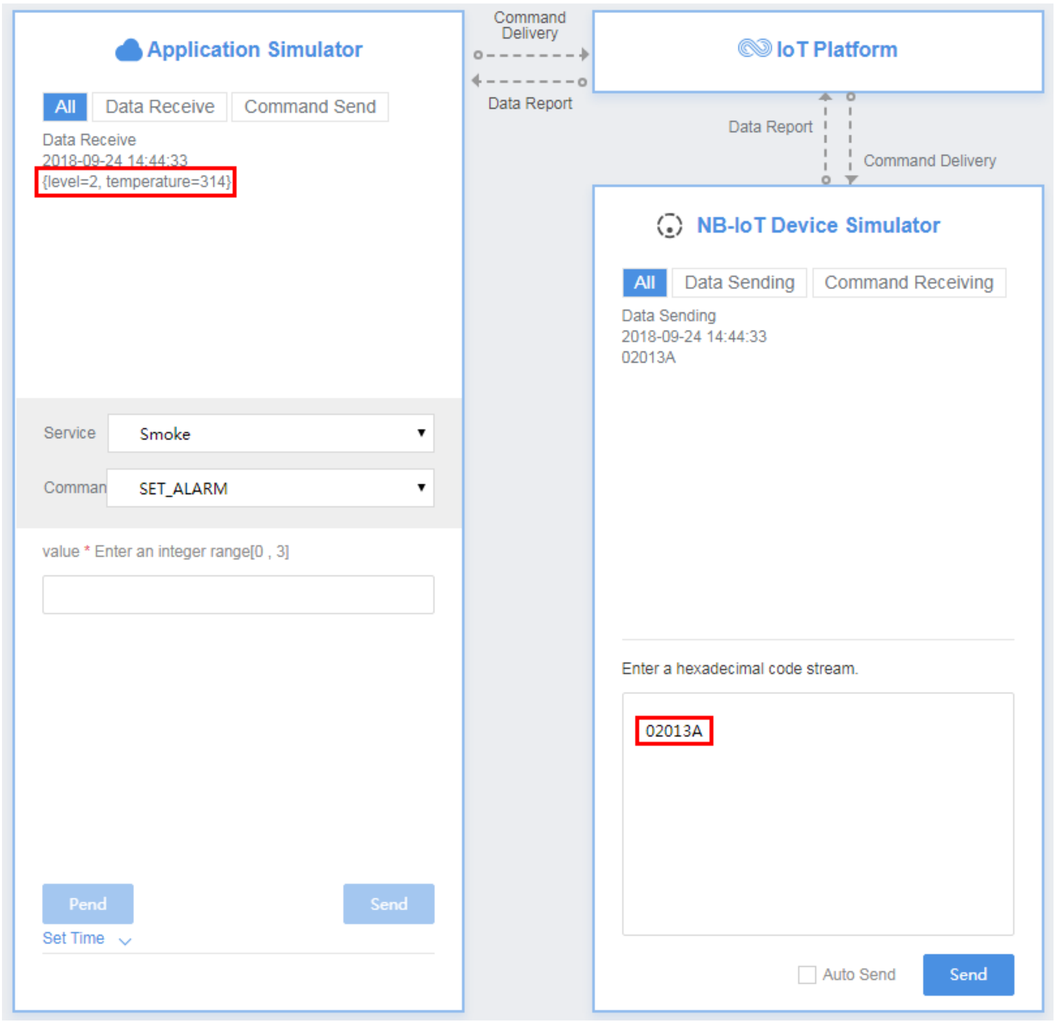


Select **No** for **Is Physical Device Available** and click **OK**.


Add Test Device
Is Physical Device Available
Yes [✓ No]
You are registering a virtual device.
OK
Cancel

Step 2 Use the device simulator to report data. For example, a hexadecimal code stream (02013A) is reported. In this code stream, **02** indicates the fire severity and its length is one byte. **013A** indicates the temperature and its length is two bytes.

View the data reporting result ({level=2, temperature=314}) in **Application Simulator**. 2 is the decimal number converted from the hexadecimal number 02 and 314 from the hexadecimal number 013A.



Data Receive
2018-09-24 14:44:33
{level=2, temperature=314}

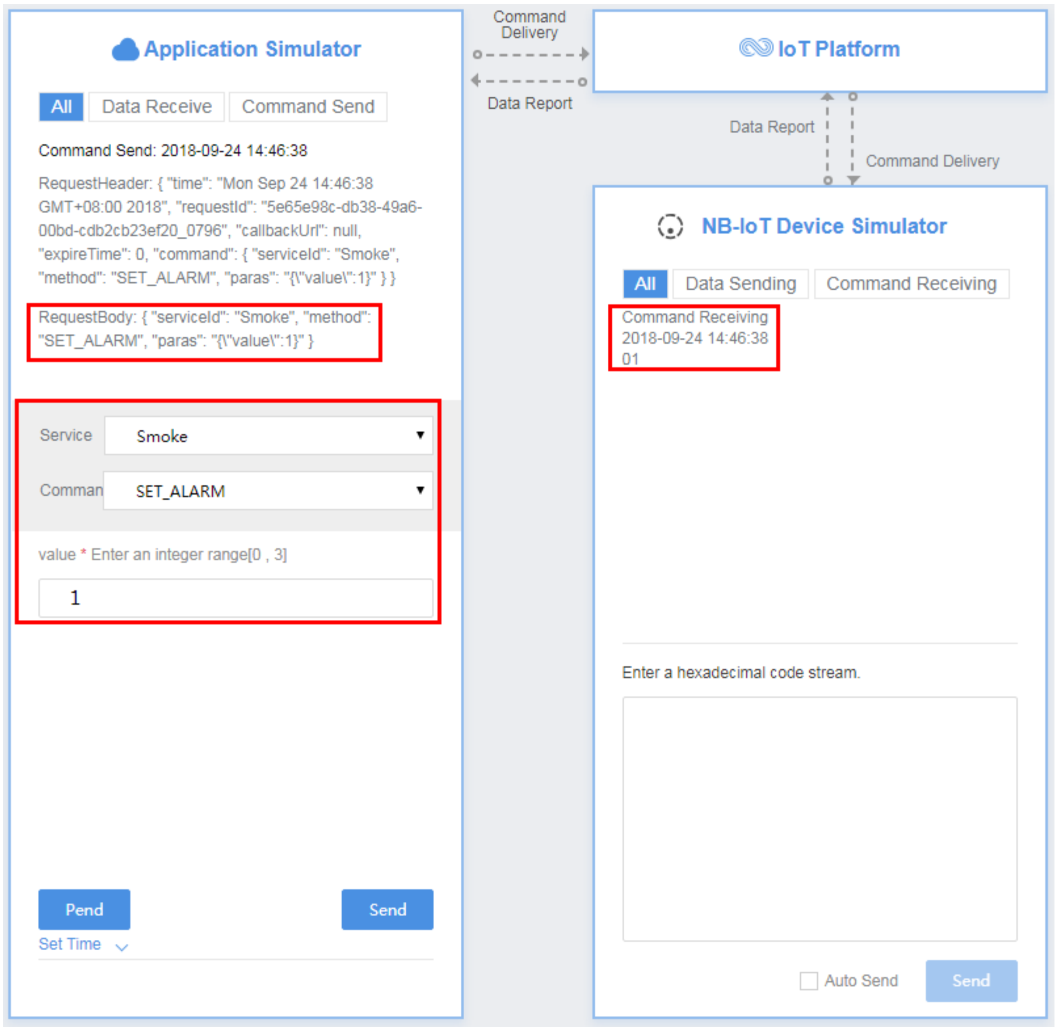
Service: Smoke
Comman: SET_ALARM
value * Enter an integer range[0 , 3]

Data Sending
2018-09-24 14:44:33
02013A

Enter a hexadecimal code stream.
02013A

Step 3 Use the application simulator to deliver a command ({ "serviceId": "Smoke", "method": "SET_ALARM", "paras": "{\"value\":1}" }).

View the command receiving result in **Device Simulator**, which is **01**. 01 is the hexadecimal number converted from the decimal number 1.



Command Send: 2018-09-24 14:46:38

RequestHeader: { "time": "Mon Sep 24 14:46:38 GMT+08:00 2018", "requestId": "5e65e98c-db38-49a6-00bd-cdb2cb23ef20_0796", "callbackUrl": null, "expireTime": 0, "command": { "serviceId": "Smoke", "method": "SET_ALARM", "paras": "{\"value\":1}" } }

RequestBody: { "serviceId": "Smoke", "method": "SET_ALARM", "paras": "{\"value\":1}" }

Service: Smoke

Comman: SET_ALARM

value * Enter an integer range[0 , 3]

1

Command Receiving
2018-09-24 14:46:38
01

Enter a hexadecimal code stream.

----End

1.4.3.2 Codec for Multiple Data Reporting Messages

Scenarios

A smoke detector provides the following functions:

- Smoke alarms (fire severity) and temperature reporting
- Remote command, which can enable the alarm function remotely

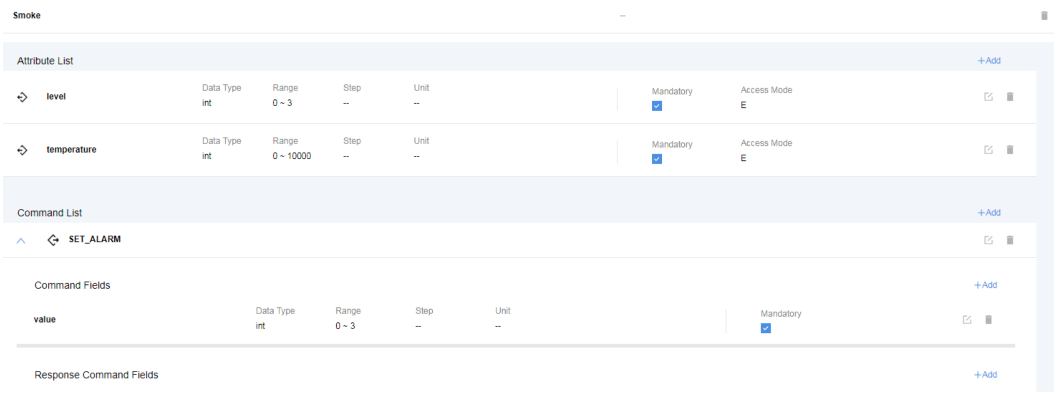
  For example, the smoke detector can report the temperature on the fire scene and remotely trigger the smoke alarm for evacuation.
- Reporting smoke alarms (fire severity) and temperature simultaneously, or reporting the temperature separately.

Defining the Profile File

Define the profile file in the development space of the smoke sensor.

- 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.


Smoke --

Attribute List +Add

	Data Type	Range	Step	Unit	Mandatory	Access Mode
level	int	0 ~ 3	--	--	✓	E
temperature	int	0 ~ 10000	--	--	✓	E

Command List +Add

SET_ALARM

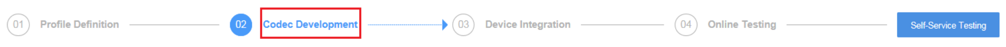
Command Fields +Add

	Data Type	Range	Step	Unit	Mandatory
value	int	0 ~ 3	--	--	✓

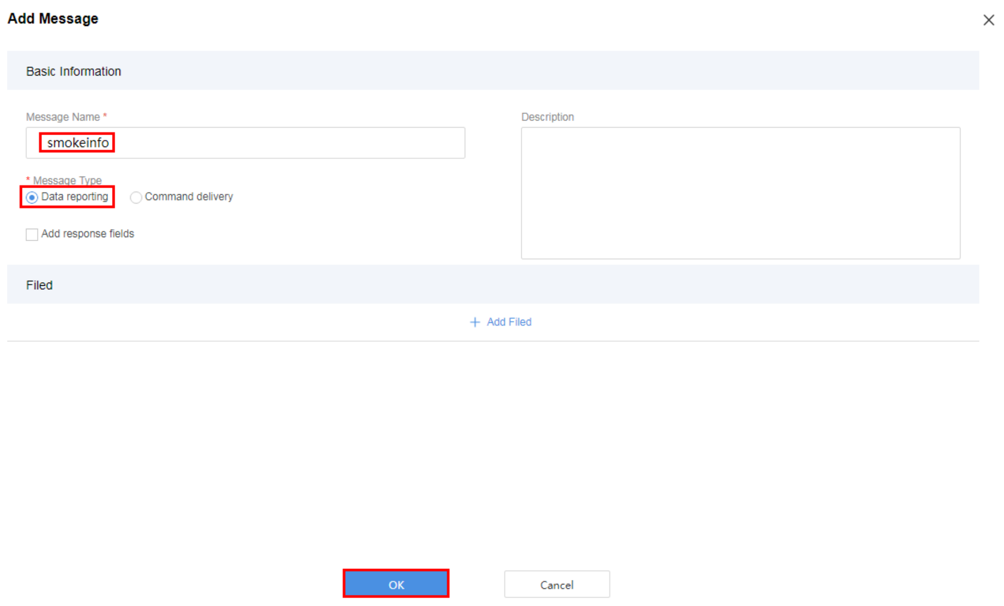
Response Command Fields +Add

Developing a Codec

Step 1 In the development space of the smoke sensor, click **Codec Development**.



Step 2 Configure a data reporting message to report the fire severity and temperature.


Add Message

Basic Information

Message Name * smokeinfo

Description

* Message Type [✓ Data reporting] Command delivery

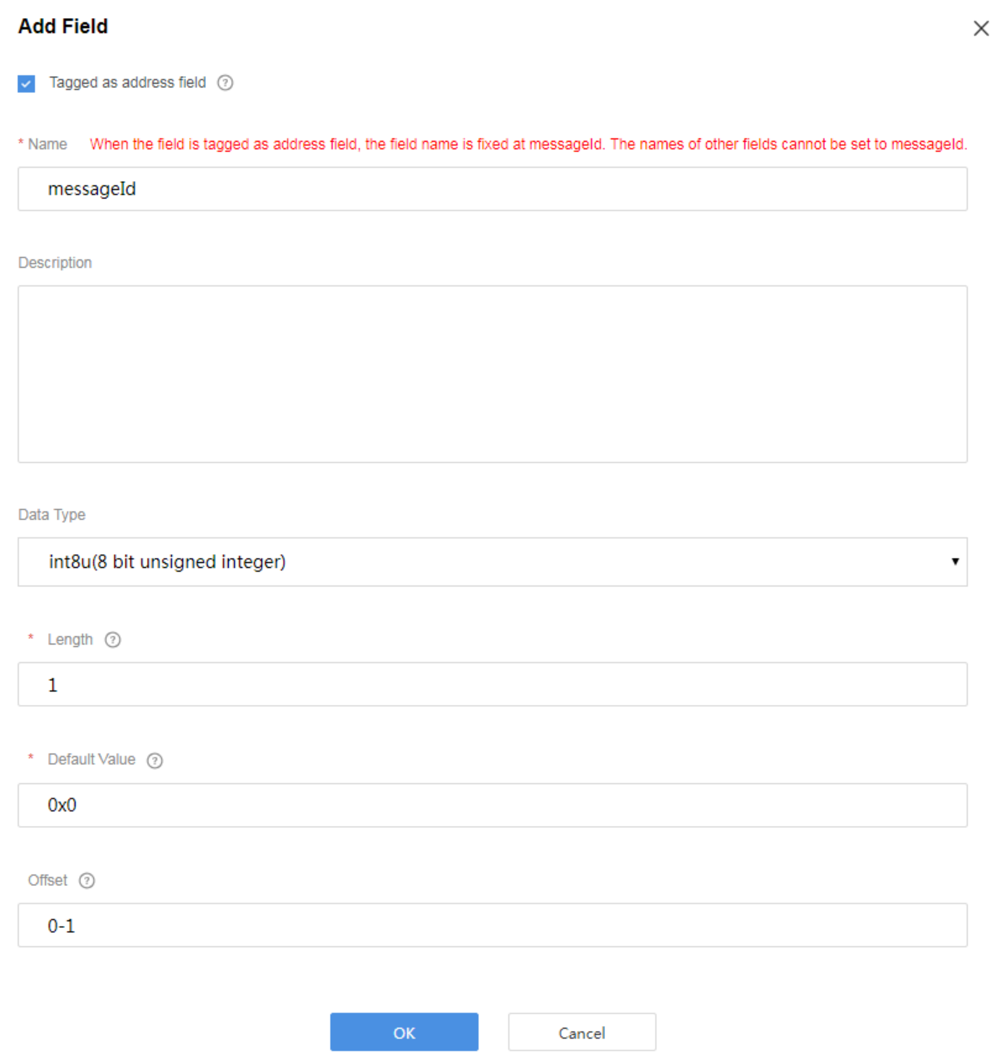
Add response fields

Filed

+ Add Filed

Add the **messageId** field to indicate the message type.

- In this scenario, there are two types of data reporting messages. Therefore, the **messageId** field must be defined to identify the message type.
- **Data Type** is configured based on the number of data reporting message types. In this scenario, only two types of data reporting messages are available. Therefore, the value **int8u** will suffice.
- **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. In this scenario, the value **0x0** is used to identify the message that reports the fire severity and temperature.


Add Field

☑ Tagged as address field

* Name When the field is tagged as address field, the field name is fixed at messageId. The names of other fields cannot be set to messageId.

messageId

Description

Data Type: int8u(8 bit unsigned integer)

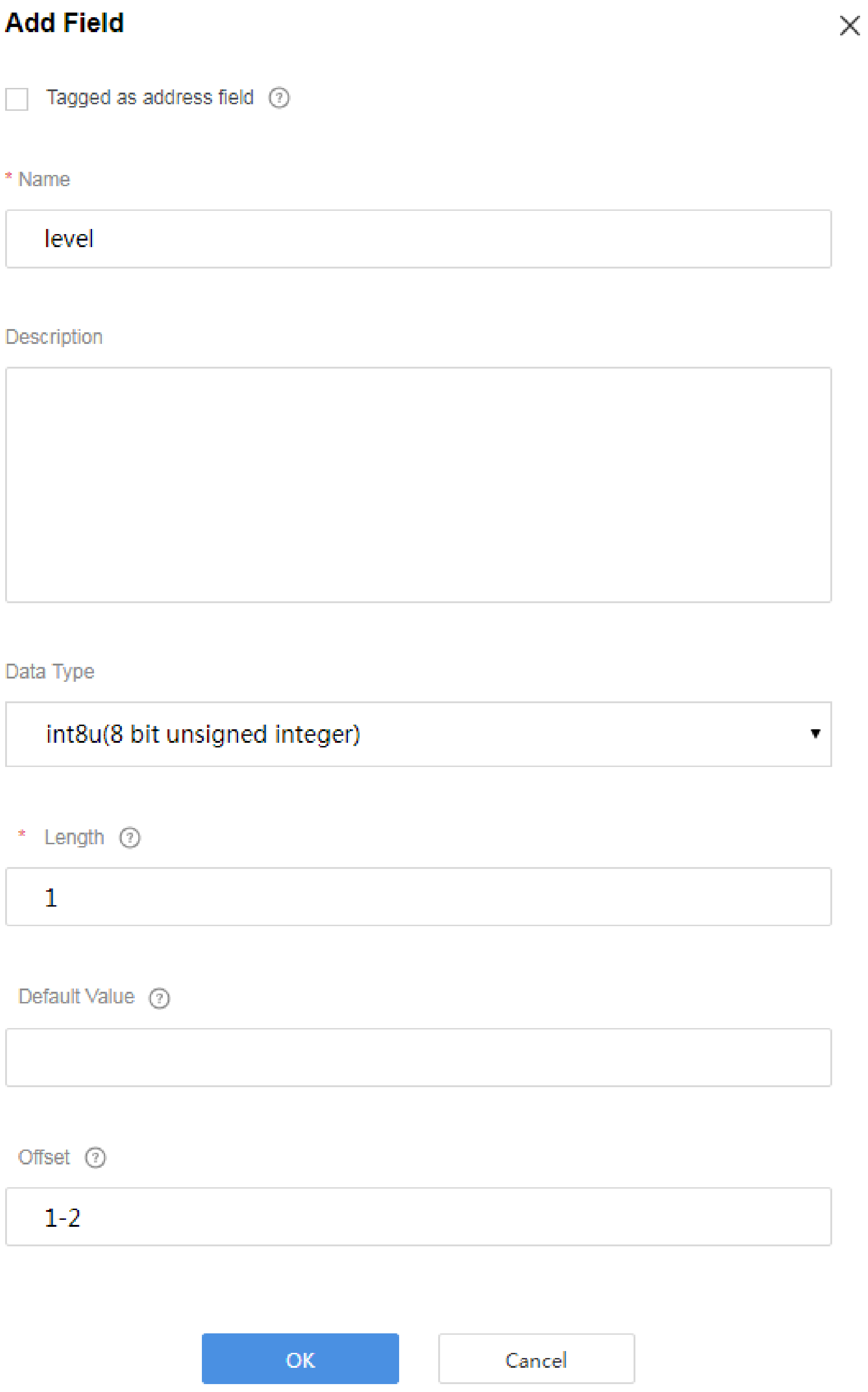
* Length: 1

* Default Value: 0x0

Offset: 0-1

Add a **level** field to indicate the fire severity.

- **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 profile file.
- The values of **Length** and **Offset** are automatically filled based on **Data Type**.



Tagged as address field ⓘ

* Name: level

Description:

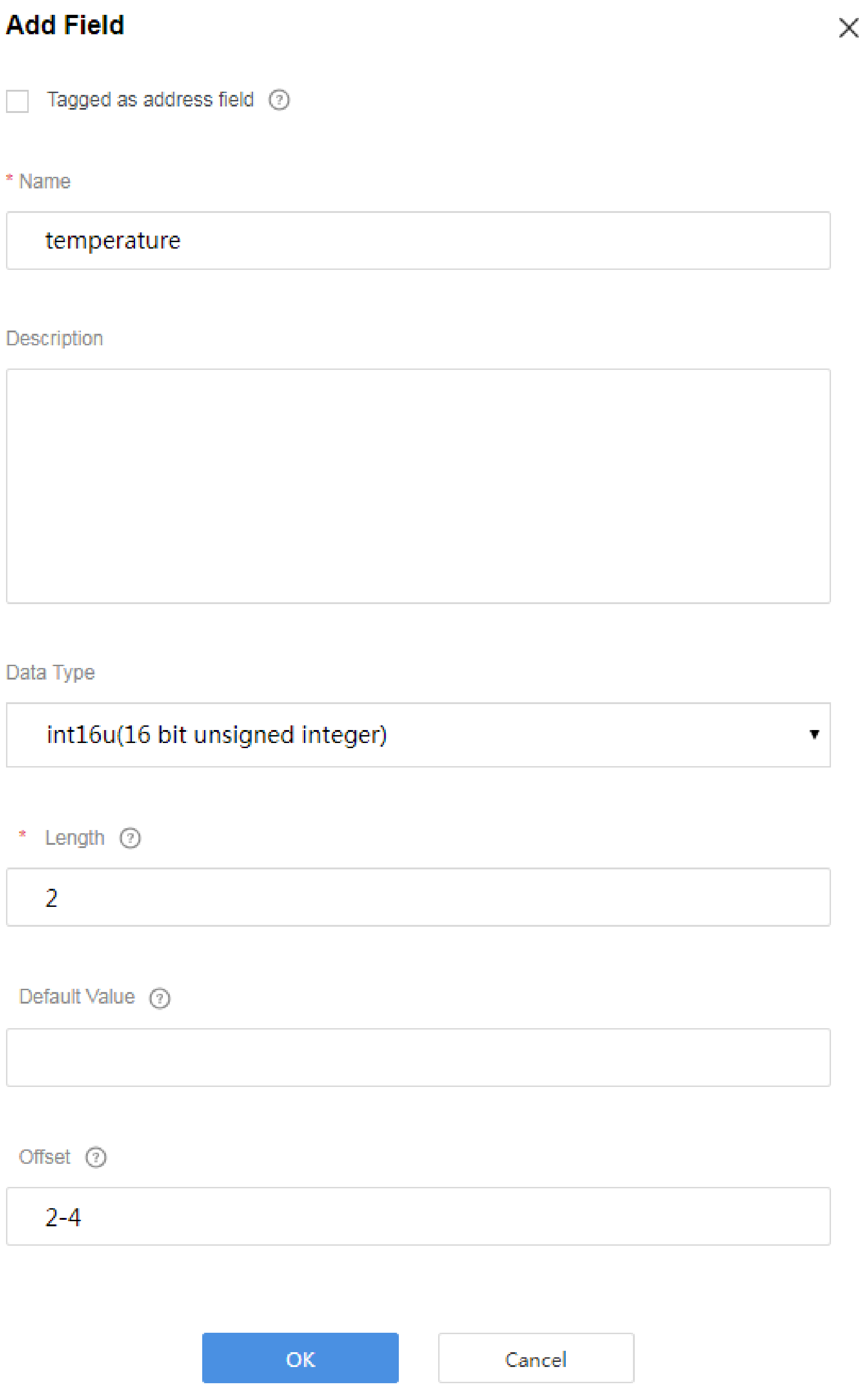
Data Type: int8u(8 bit unsigned integer)

* Length: 1

Default Value:

Offset: 1-2

Add the **temperature** field to indicate the temperature at the fire scene. In the profile file, the maximum value of **temperature** is **1000**. Therefore, set the data type of the **temperature** field to **int16u** in the codec to meet the value range requirement of **temperature**.



Tagged as address field

* Name: temperature

Description:

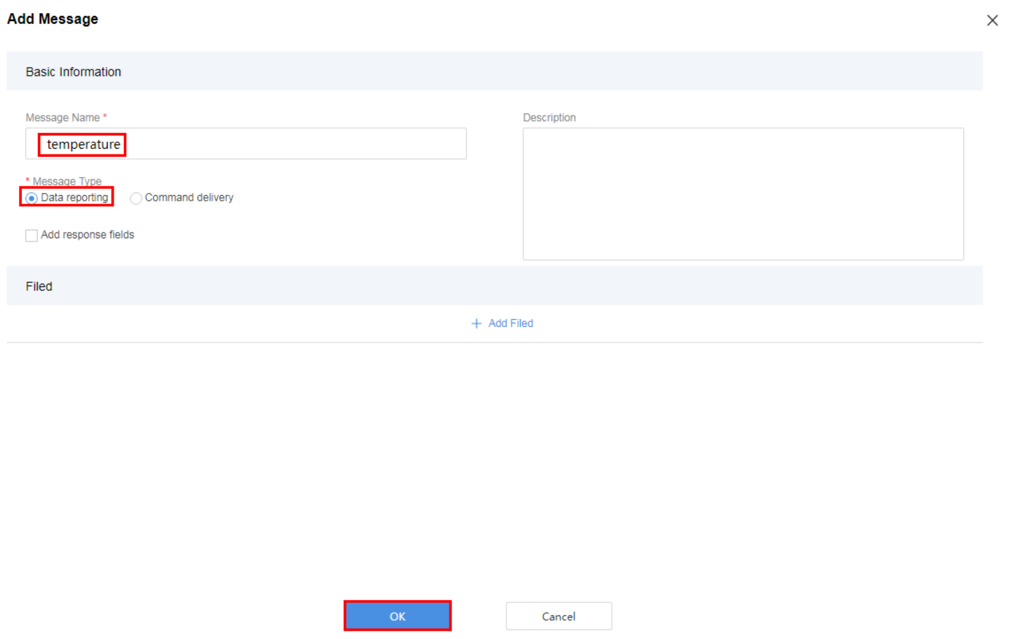
Data Type: int16u(16 bit unsigned integer)

* Length: 2

Default Value:

Offset: 2-4

Step 3 Configure a data reporting message to report only the temperature.


Add Message

Basic Information

Message Name *: temperature

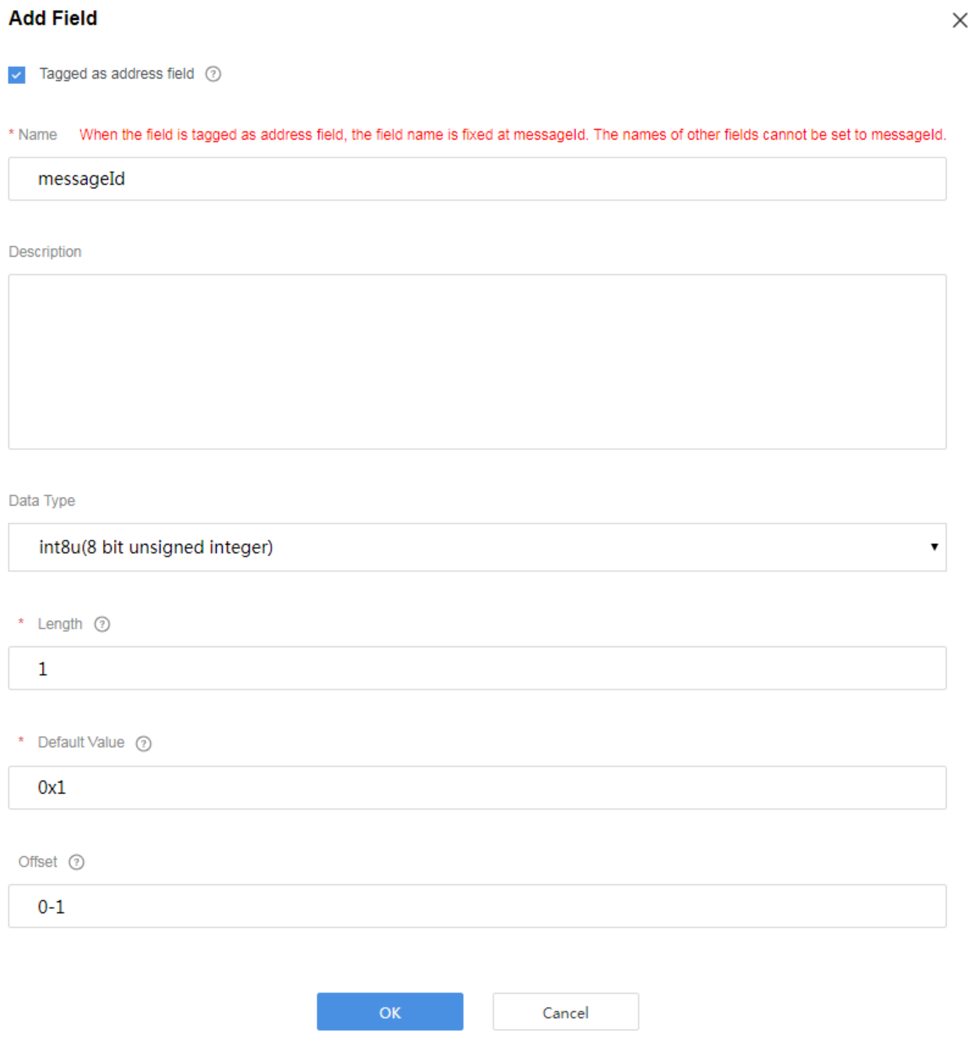
Description

* Message Type: [✓ Data reporting] Command delivery

Add response fields

Filed

Add the **messageId** field to indicate the message type. In this scenario, the value **0x1** is used to identify the message that reports only the temperature.


Add Field

[✓] Tagged as address field

* Name When the field is tagged as address field, the field name is fixed at messageId. The names of other fields cannot be set to messageId.

messageId

Description

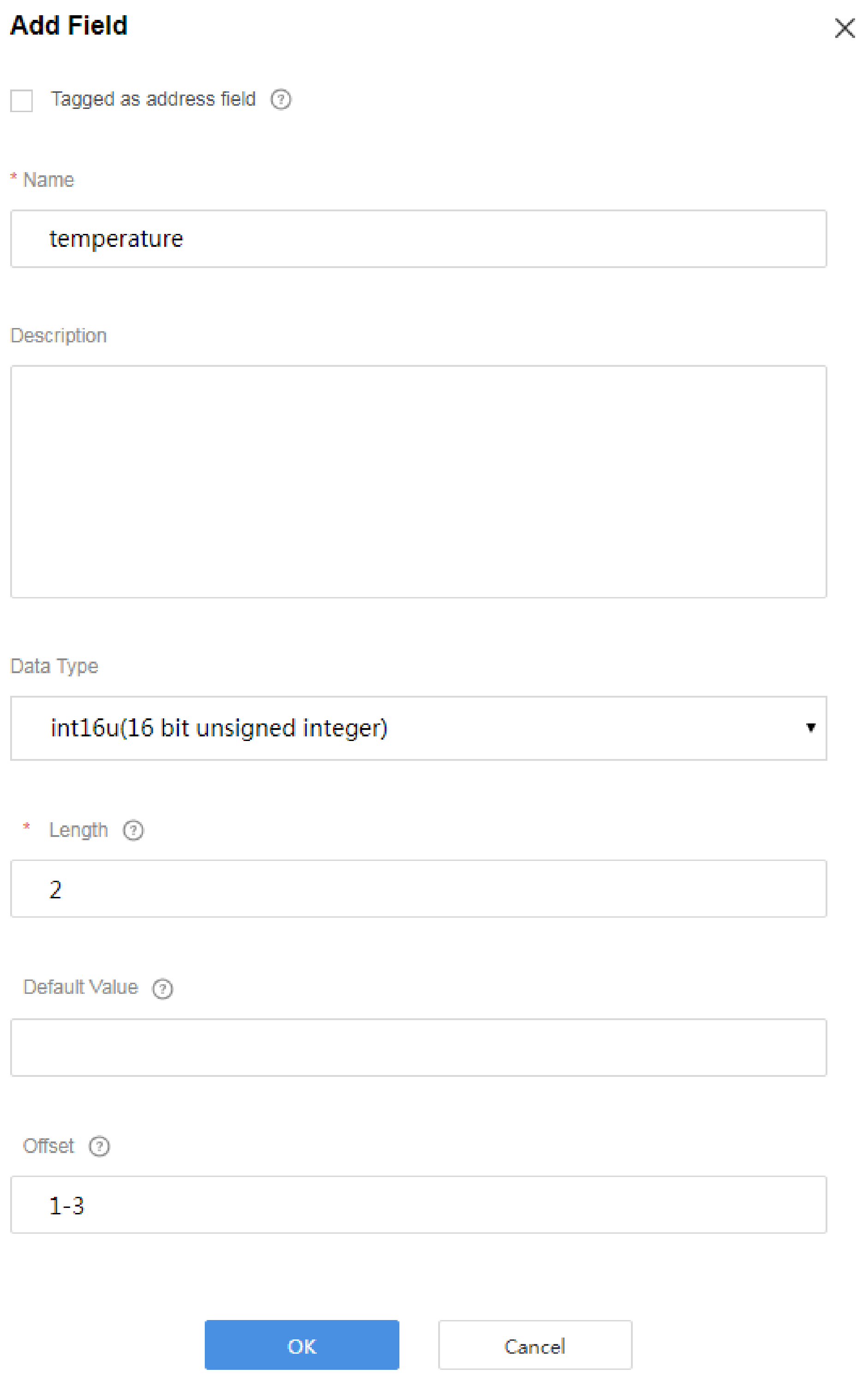
Data Type: int8u(8 bit unsigned integer)

* Length: 1

* Default Value: 0x1

Offset: 0-1

Add the **temperature** field to indicate the temperature at the fire scene.


Add Field

Tagged as address field

* Name

temperature

Description

Data Type

int16u(16 bit unsigned integer)

* Length

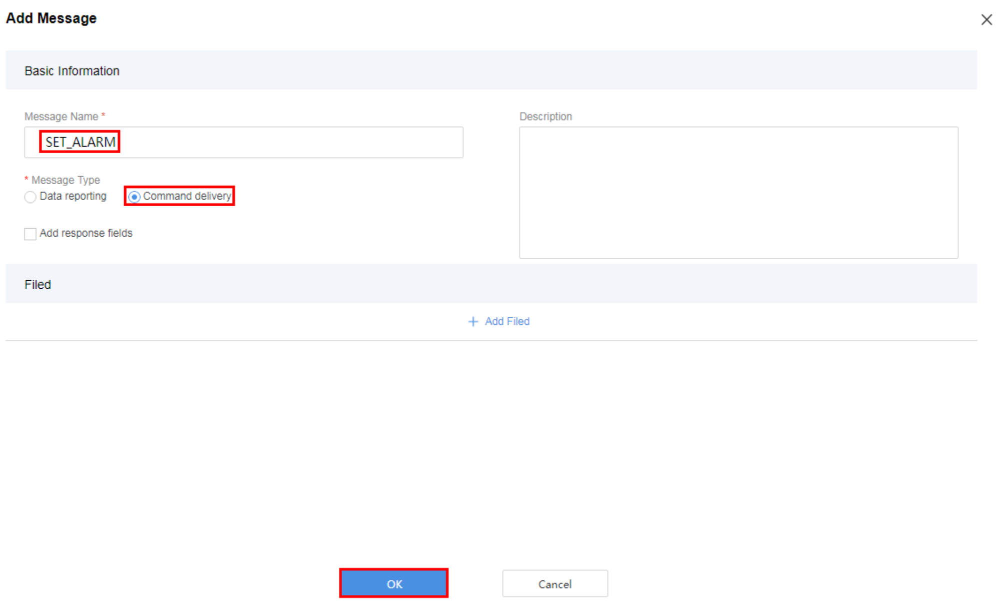
2

Default Value

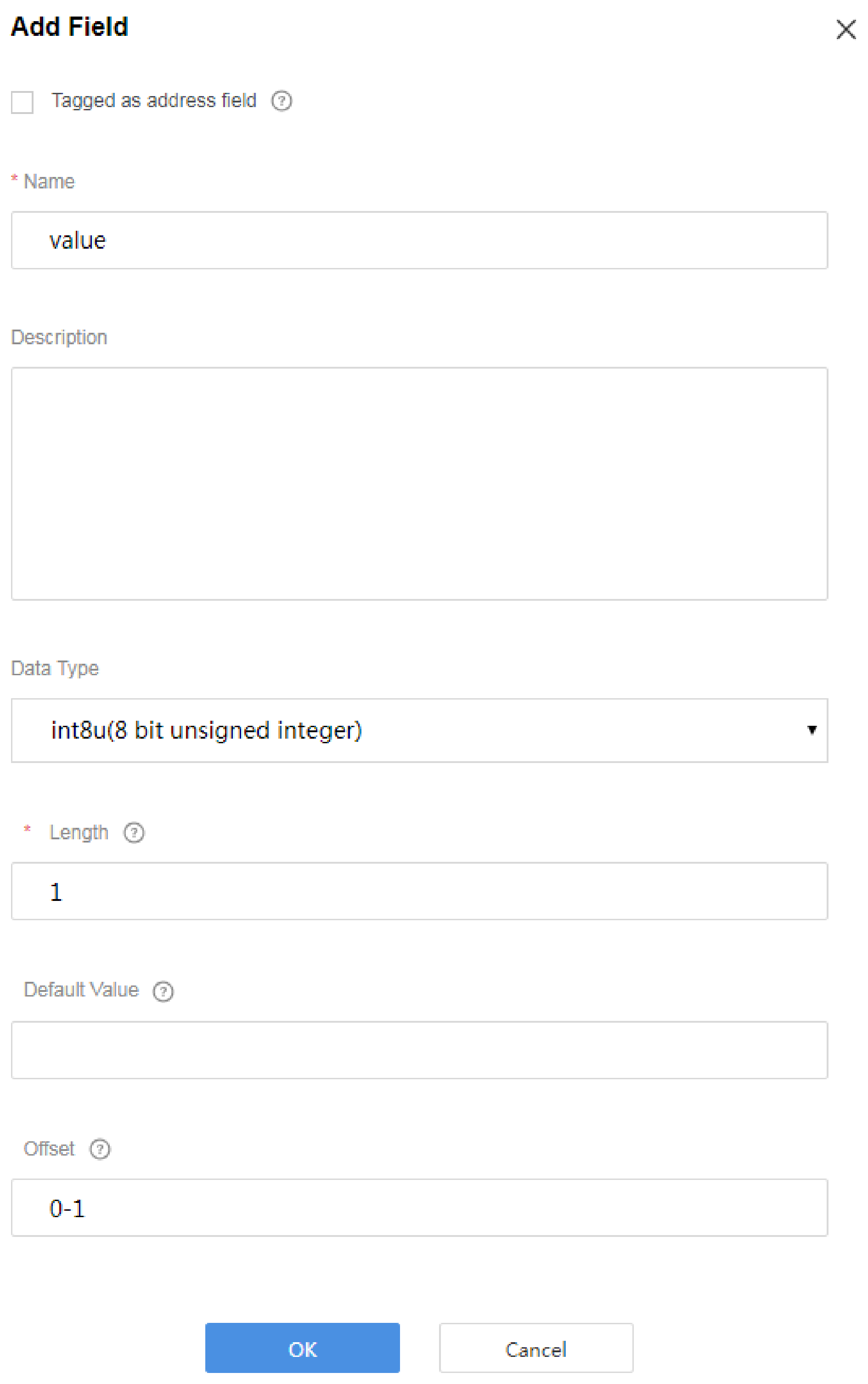
Offset

1-3

Step 4 Configure a command delivery message.



Add the **value** field to indicate the parameter value of the delivered command.



Tagged as address field

* Name: value

Description:

Data Type: int8u(8 bit unsigned integer)

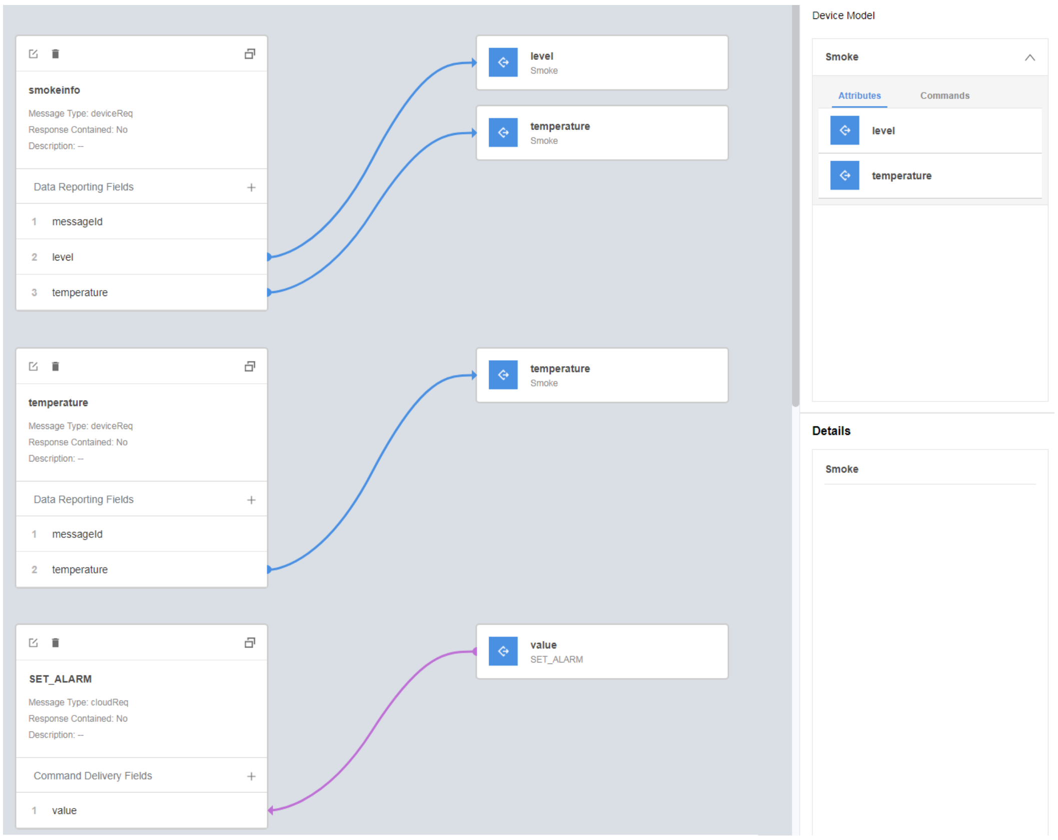
* Length: 1

Default Value:

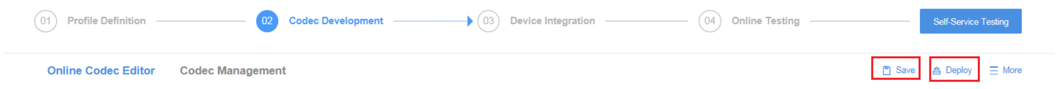
Offset: 0-1

Step 5 Drag the property fields and command fields in **Device Model** on the right to set up a mapping with the fields in the data reporting message and command delivery message.

The **level** and **temperature** fields are mapped to the corresponding properties in the profile file. The **messageId** field is used to identify message types and does not need to be mapped.



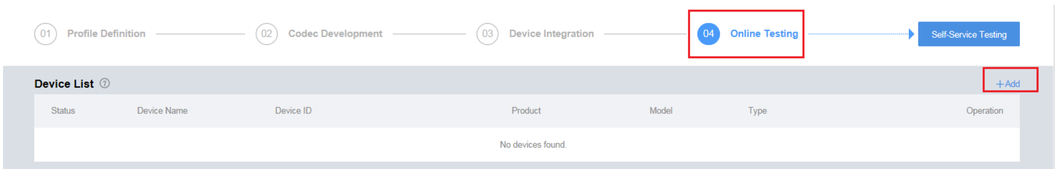
Step 6 Click **Save** and then **Deploy** to deploy the codec on the IoT platform.



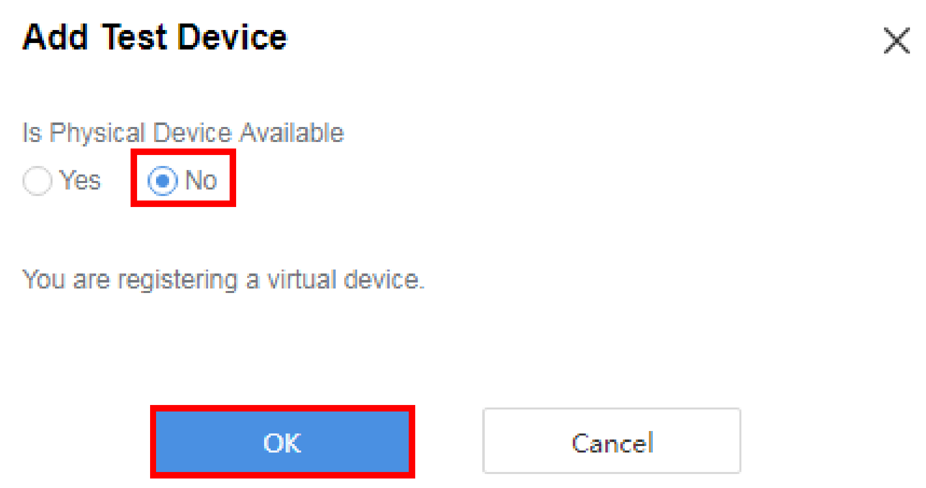
----End

Testing the Codec

Step 1 In the development space of the smoke sensor, click **Online Testing** and add a virtual device to test the codec.



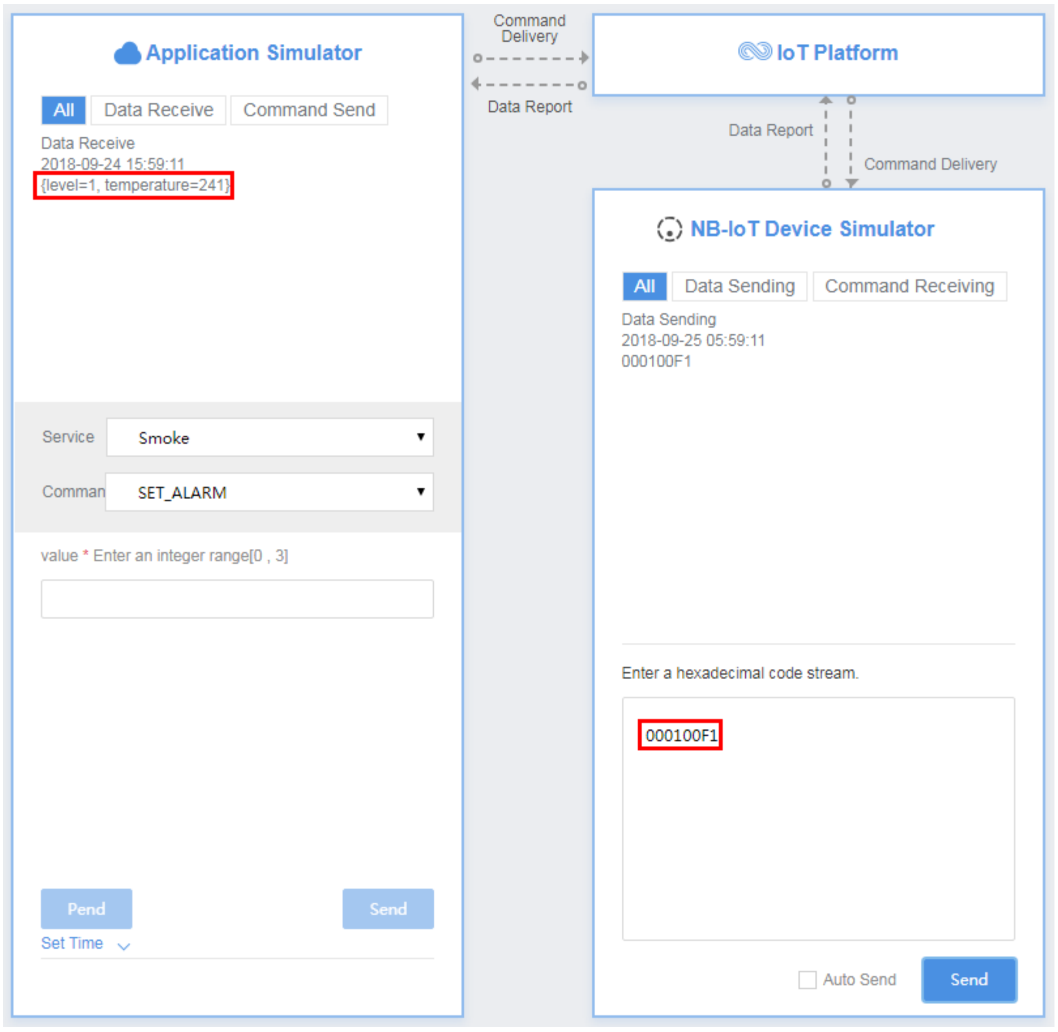
Select **No** for **Is Physical Device Available** and click **OK**.


Add Test Device
Is Physical Device Available
Yes [✓ No]
You are registering a virtual device.
OK
Cancel

Step 2 Use the device simulator to report data.

For example, a hexadecimal code stream (000100F1) is reported. In this code stream, **00** indicates the **messageId** field and specifies that this message reports the fire severity and temperature. **01** indicates the fire severity and its length is one byte. **00F1** indicates the temperature and its length is two bytes.

View the data reporting result ({level=1, temperature=241}) in **Application Simulator**. 1 is the decimal number converted from the hexadecimal number 01 and 241 from the hexadecimal number 00F1.



Data Receive
2018-09-24 15:59:11
{level=1, temperature=241}

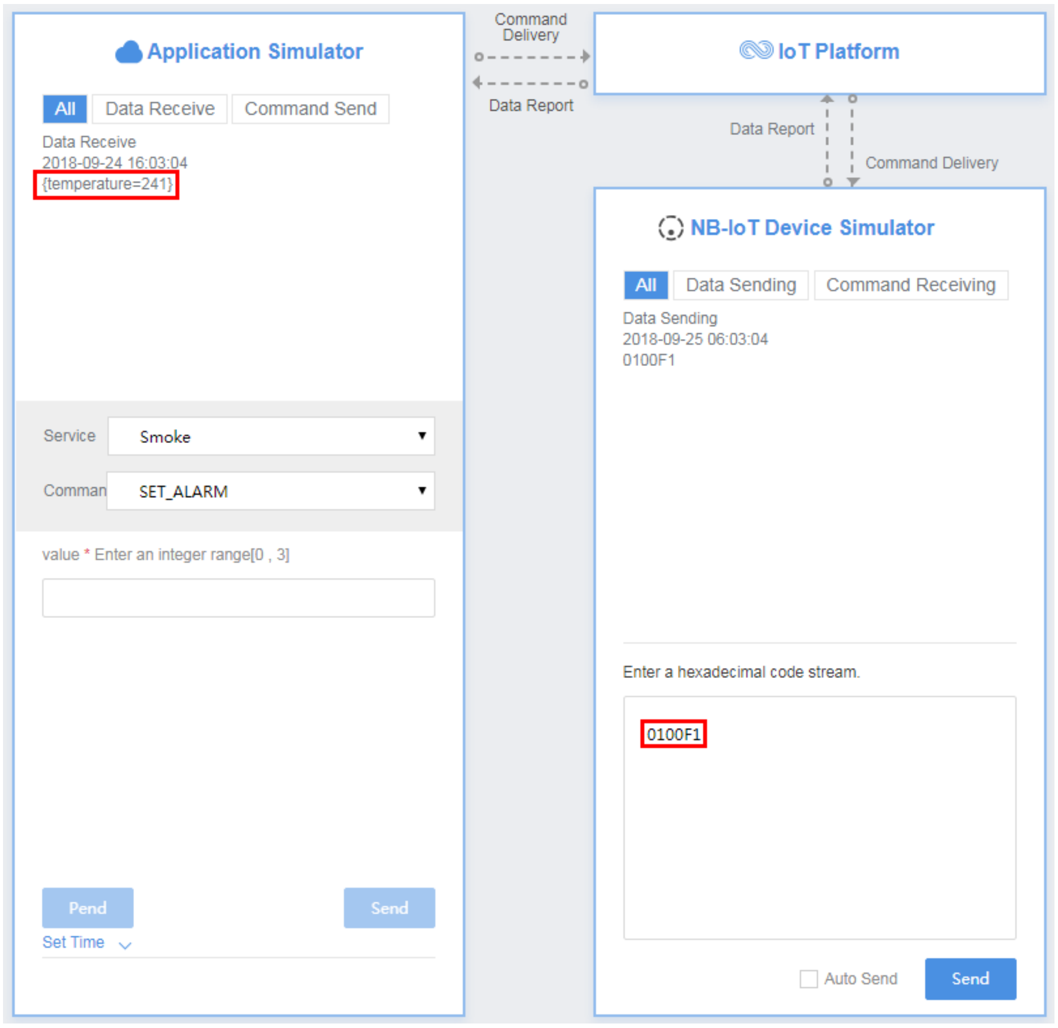
Service: Smoke
Comman: SET_ALARM
value * Enter an integer range[0 , 3]

Data Sending
2018-09-25 05:59:11
000100F1

Enter a hexadecimal code stream.
000100F1

Take another hexadecimal code stream (0100F1) as an example. **01** indicates the **messageId** field and specifies that this message reports only the temperature. **00F1** indicates the temperature and its length is two bytes.

View the data reporting result ({temperature=241}) in **Application Simulator**. 241 is the decimal number converted from the hexadecimal number 00F1.



Data Receive
2018-09-24 16:03:04
{temperature=241}

Service: Smoke
Comman: SET_ALARM
value * Enter an integer range[0 , 3]

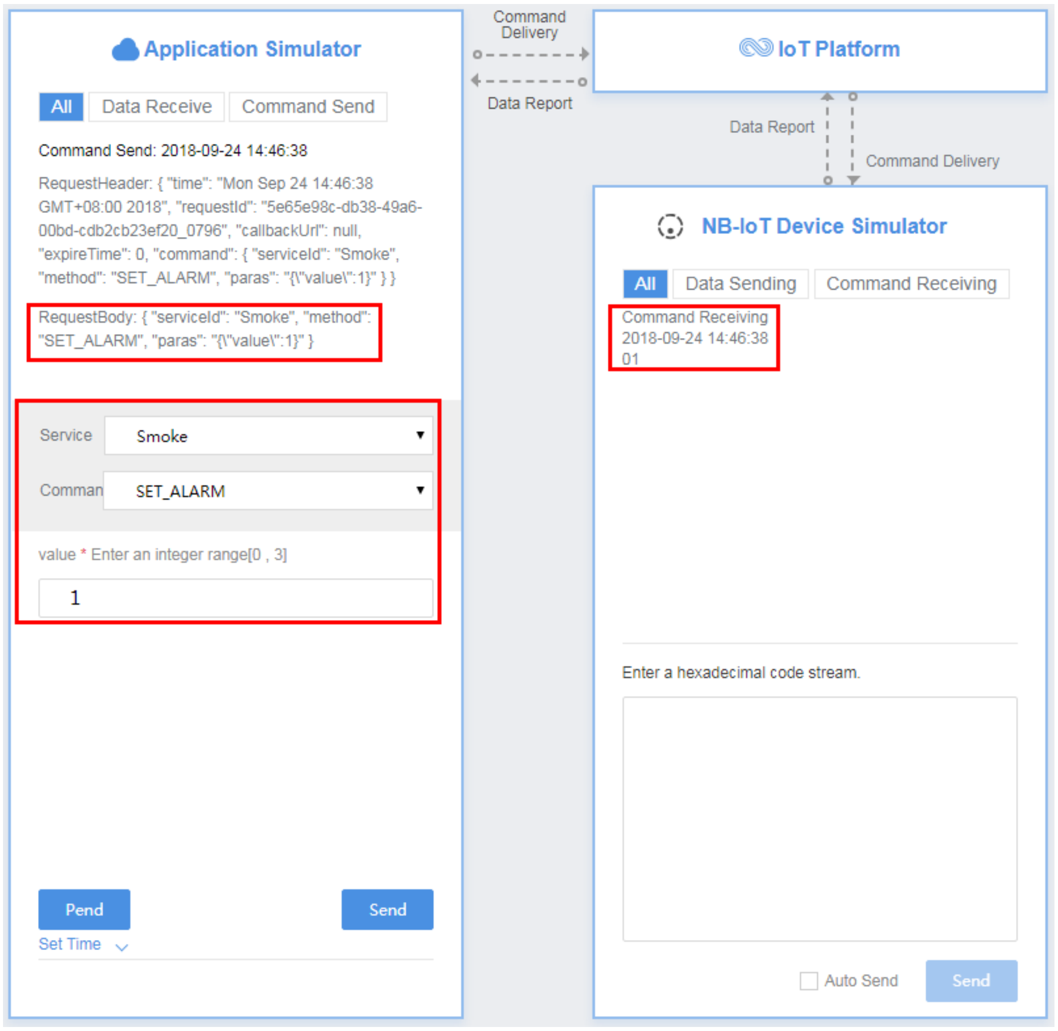
Data Sending
2018-09-25 06:03:04
0100F1

Enter a hexadecimal code stream.

0100F1

Step 3 Use the application simulator to deliver a command ({ "serviceId": "Smoke", "method": "SET_ALARM", "paras": "{\"value\":1}" }).

View the command receiving result in **Device Simulator**, which is **01**. 01 is the hexadecimal number converted from the decimal number 1.



Command Send: 2018-09-24 14:46:38

RequestHeader: { "time": "Mon Sep 24 14:46:38 GMT+08:00 2018", "requestId": "5e65e98c-db38-49a6-00bd-cdb2cb23ef20_0796", "callbackUrl": null, "expireTime": 0, "command": { "serviceId": "Smoke", "method": "SET_ALARM", "paras": "{\"value\":1}" } }

RequestBody: { "serviceId": "Smoke", "method": "SET_ALARM", "paras": "{\"value\":1}" }

Service: Smoke

Comman: SET_ALARM

value * Enter an integer range[0 , 3]

1

Command Receiving
2018-09-24 14:46:38
01

Enter a hexadecimal code stream.

----End

1.4.3.3 Codec for Strings and Variable-Length Strings

Scenarios

A smoke detector provides the following functions:

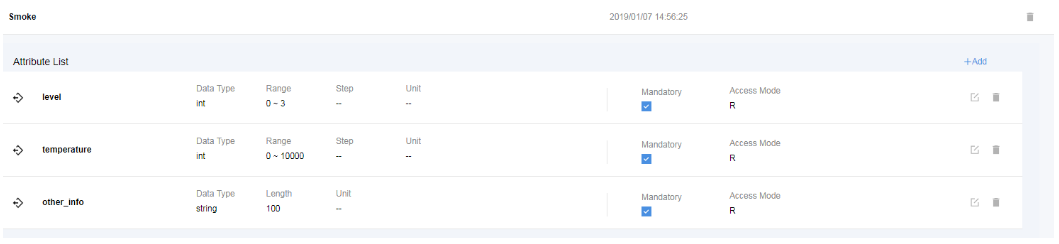
- Reporting smoke alarms (fire severity) and temperature simultaneously, or reporting the temperature separately.
- Reporting description. The data type of description can be **string (string type)** or **varstring (variable-length string type)**.

  NOTE

  This scenario describes how to develop a codec for data in strings and data in variable-length strings. The data reporting and command delivery codecs are developed in the same way. Therefore, data reporting is used as an example and command delivery is not described.

Defining the Profile File

Define the profile file in the development space of the smoke sensor.


Smoke 2019/01/07 14:56:25

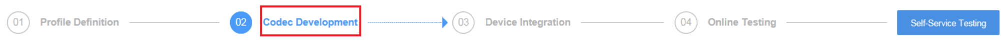
Attribute List +Add

level	Data Type: int	Range: 0 ~ 3	Step: --	Unit: --	Mandatory: ✓	Access Mode: R
temperature	Data Type: int	Range: 0 ~ 10000	Step: --	Unit: --	Mandatory: ✓	Access Mode: R
other_info	Data Type: string	Length: 100	Unit: --		Mandatory: ✓	Access Mode: R

Developing a Codec

This section describes only the procedure for developing the codec for reporting the description (**other_info**). For details on how to develop the codec for reporting the smoke alarms (**level**) and temperature (**temperature**), see **Codec for Multiple Data Reporting Messages**.

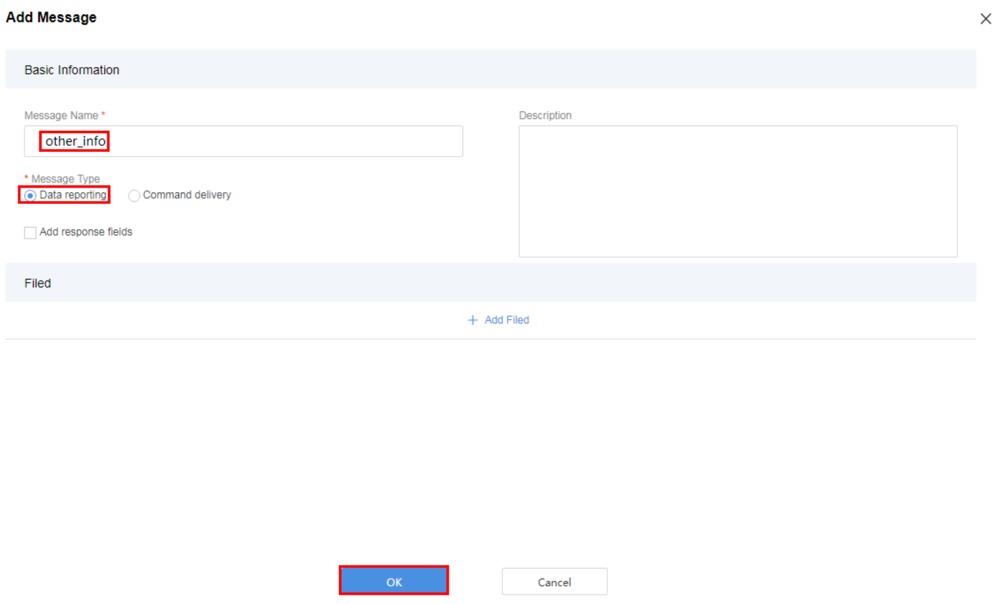
Step 1 In the development space of the smoke sensor, click **Codec Development**.



Step 2 Configure a data reporting message to report the fire severity and temperature. For details, see **Step 2**.

Step 3 Configure a data reporting message to report only the temperature. For details, see **Step 3**.

Step 4 Configure a data reporting message to report the description of the string type.


Add Message

Basic Information

Message Name *: other_info

Description:

* Message Type: [✓ Data reporting] Command delivery

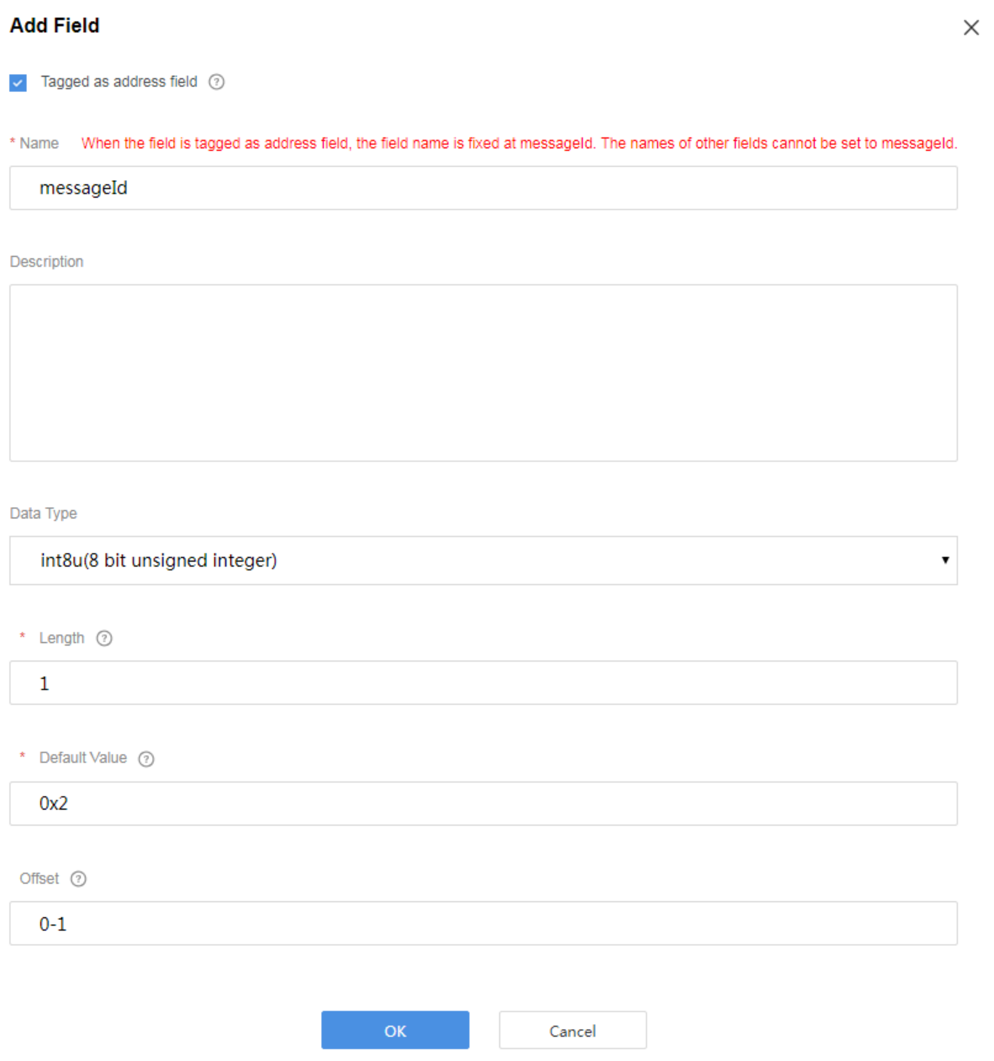
Add response fields

Filed

+ Add Filed

OK Cancel

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 **0x2** is used to identify the message that reports the description (of the string type).



☑ Tagged as address field

* Name When the field is tagged as address field, the field name is fixed at messageId. The names of other fields cannot be set to messageId.

messageId

Description

Data Type

int8u(8 bit unsigned integer)

* Length

1

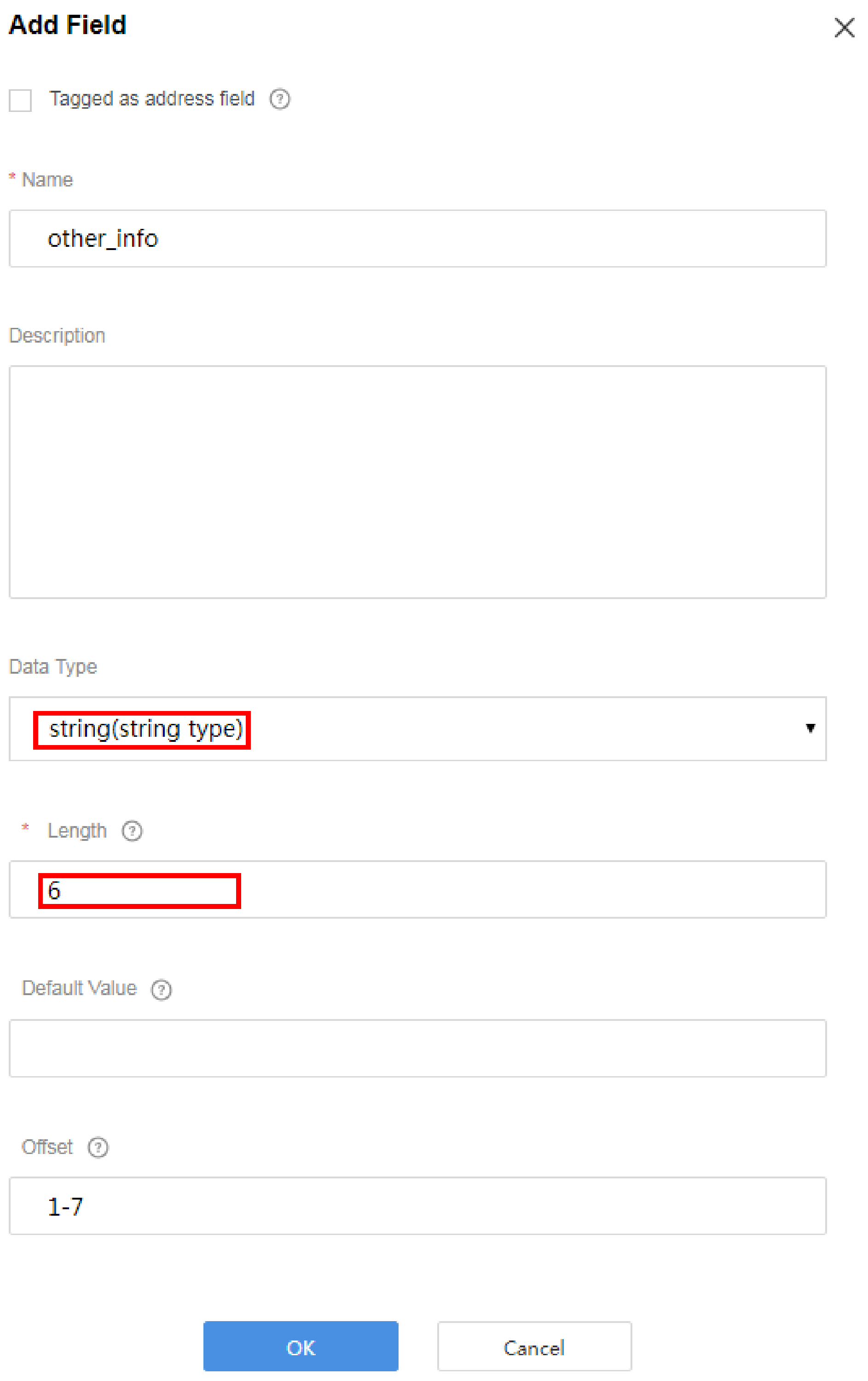
* Default Value

0x2

Offset

0-1

Add the **other_info** field to indicate the description of the string type. In this scenario, set **Length** to **6**.


Add Field

Tagged as address field

* Name

other_info

Description

Data Type

string(string type)

* Length

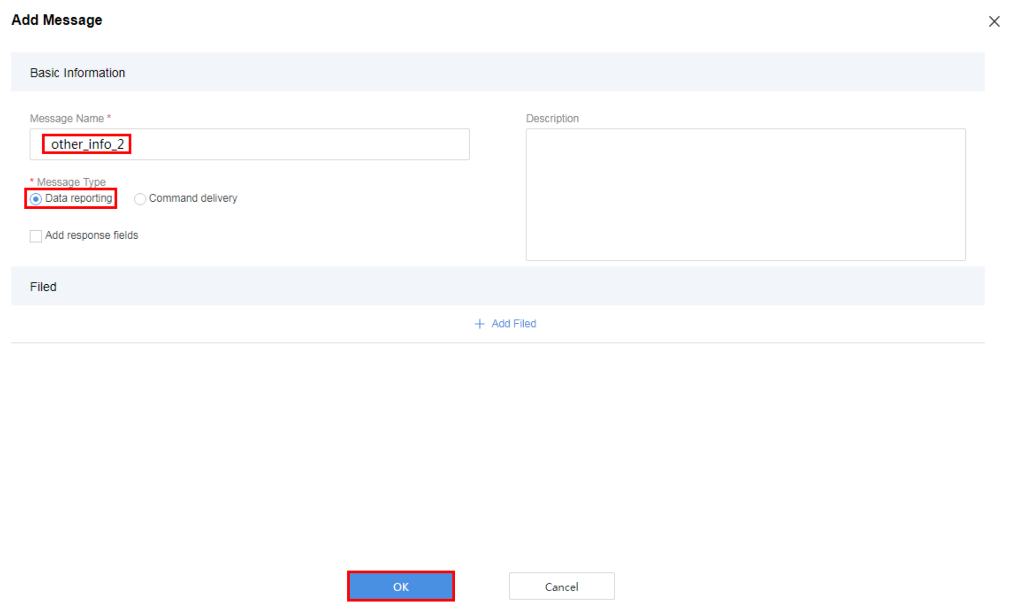
6

Default Value

Offset

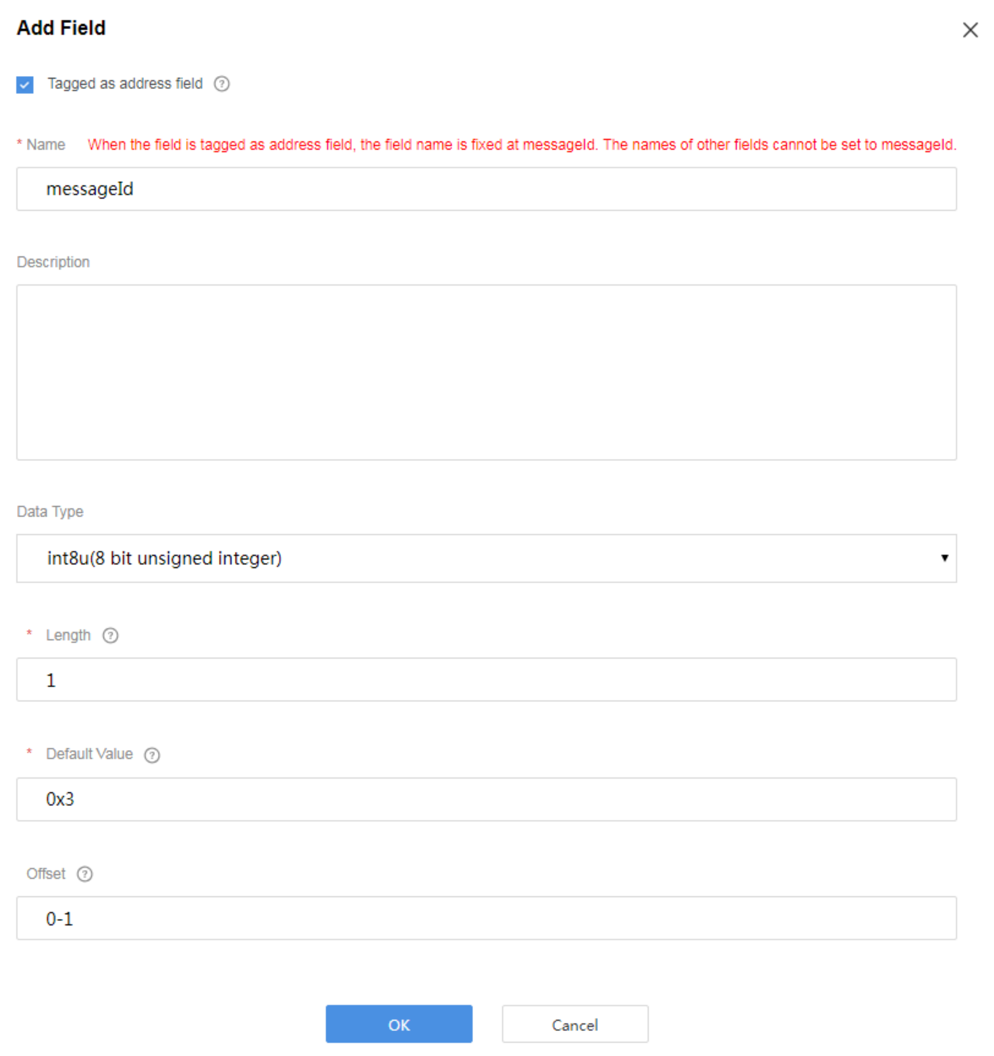
1-7

Step 5 Configure a data reporting message to report the description of the variable-length string type.



Message Name *: other_info_2
* Message Type: [✓ Data reporting] Command delivery
Add response fields

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).


Add Field

☑ Tagged as address field

* Name When the field is tagged as address field, the field name is fixed at messageId. The names of other fields cannot be set to messageId.

messageId

Description

Data Type

int8u(8 bit unsigned integer)

* Length

1

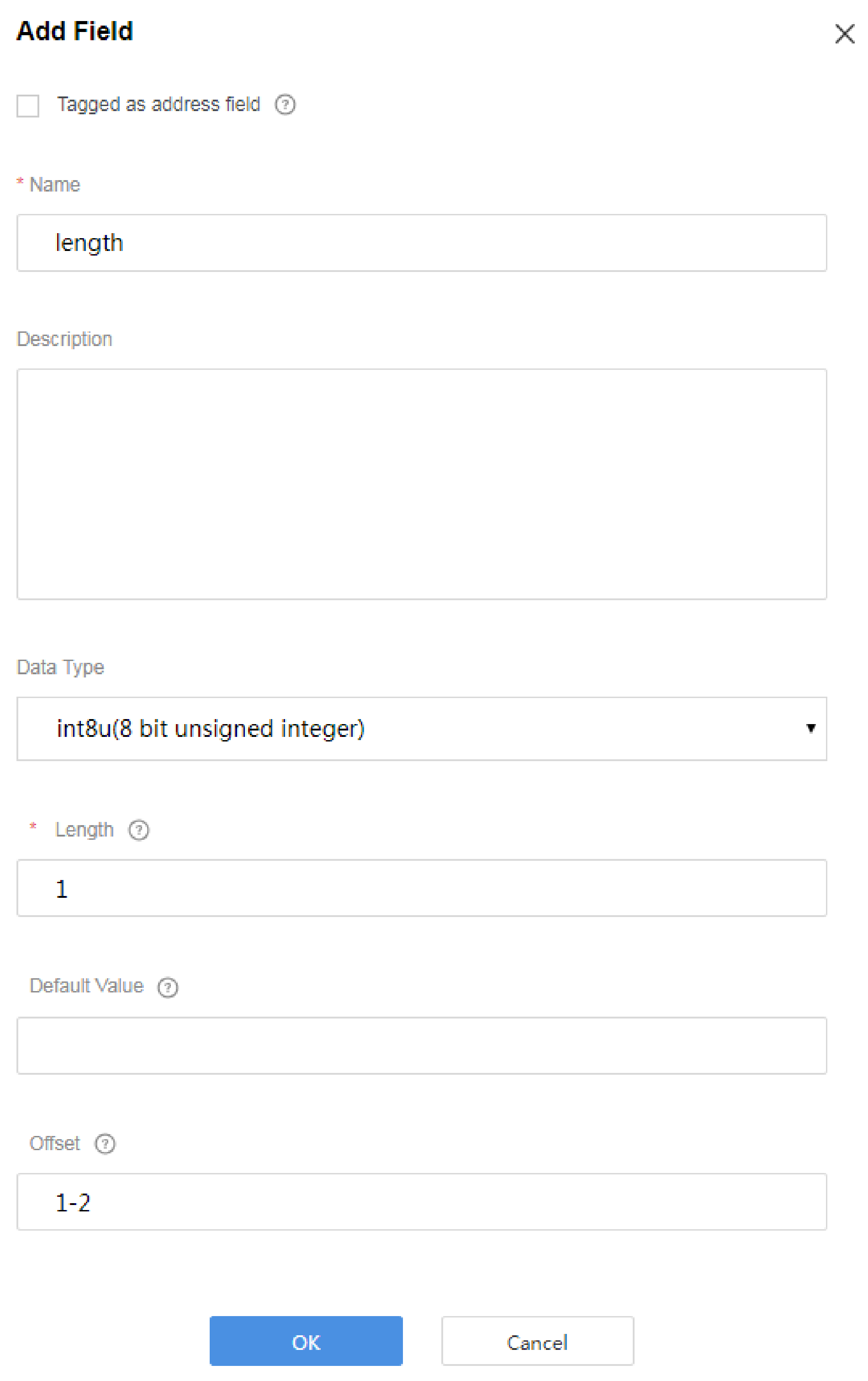
* Default Value

0x3

Offset

0-1

Add the **length** field to indicate the length of a string. **Data Type** is configured based on the length of the variable-length string. If the string contains 255 or fewer characters, set this parameter to **int8u**.


Add Field

☐ Tagged as address field

* Name

length

Description

Data Type

int8u(8 bit unsigned integer)

* Length

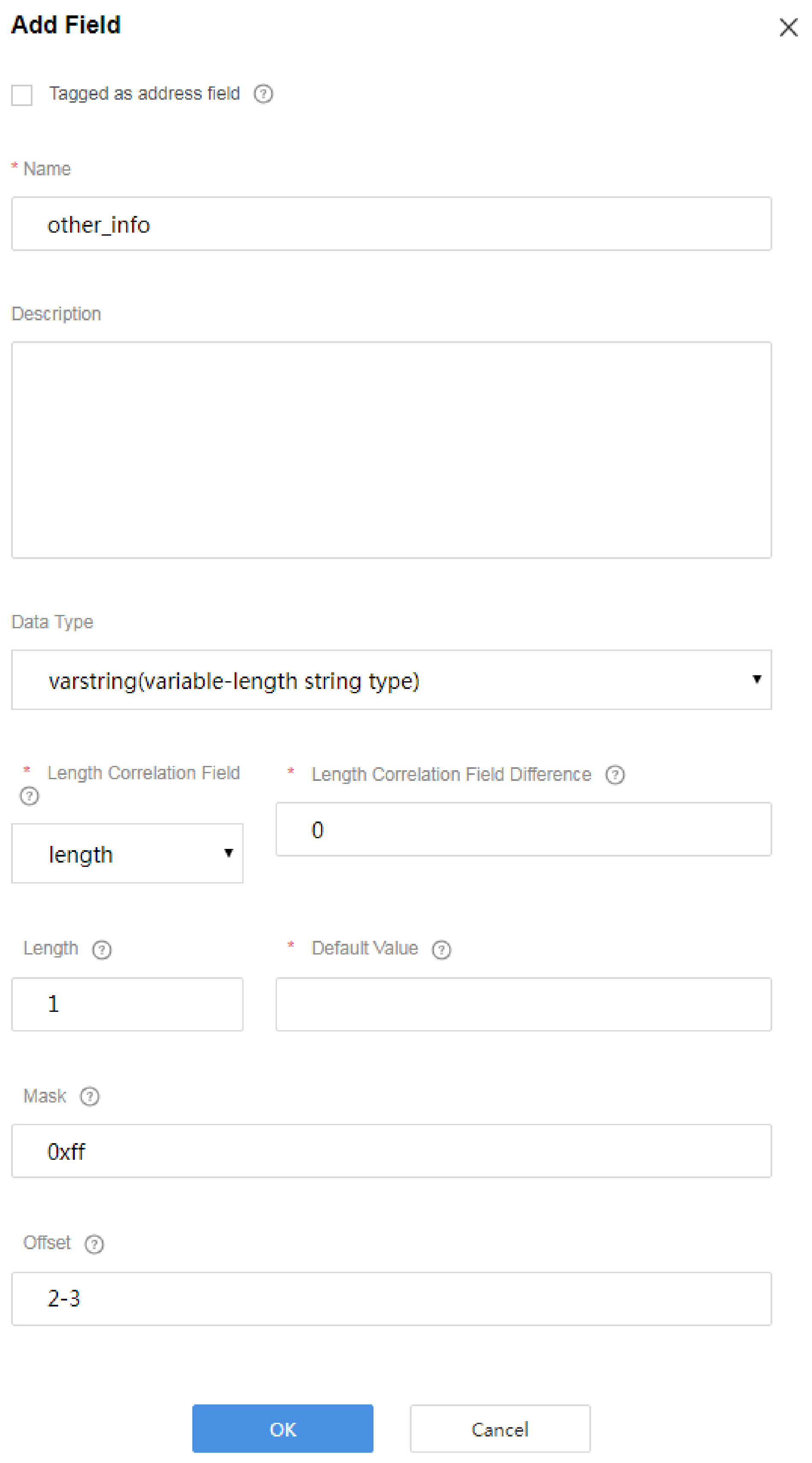
1

Default Value

Offset

1-2

Add the **other_info** field to indicate 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.


Add Field

☐ Tagged as address field

* Name

other_info

Description

Data Type

varstring(variable-length string type)

* Length Correlation Field

length

* Length Correlation Field Difference

0

Length

1

* Default Value

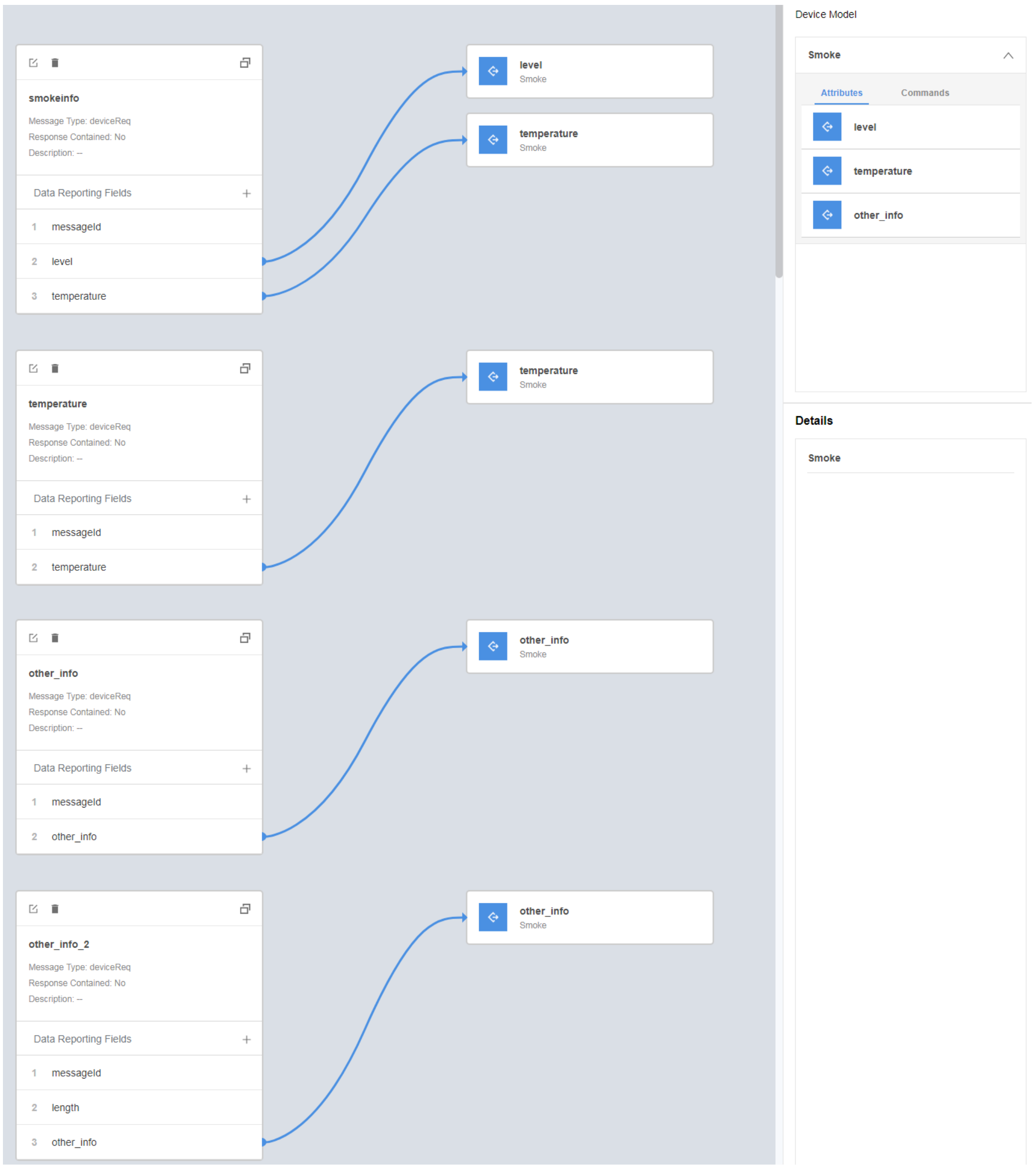
Mask

0xff

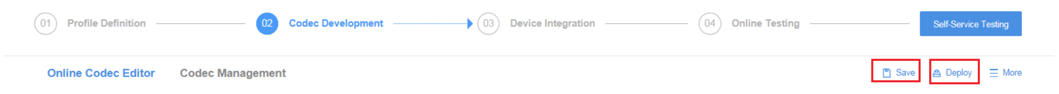
Offset

2-3

Step 6 Drag the property fields in **Device Model** on the right to set up a mapping with the fields in the data reporting messages.



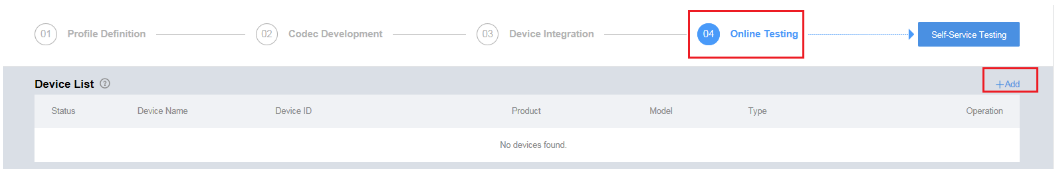
Step 7 Click **Save** and then **Deploy** to deploy the codec on the IoT platform.



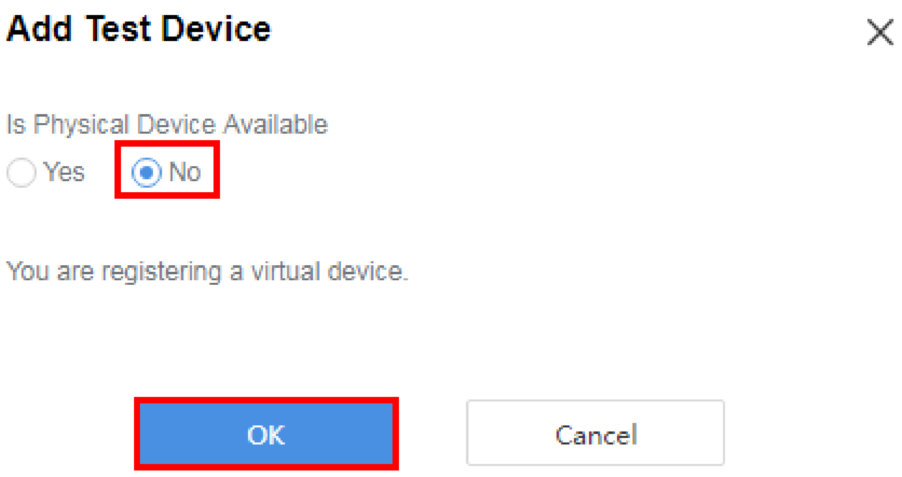
----End

Testing the Codec

Step 1 In the development space of the smoke sensor, click **Online Testing** and add a virtual device to test the codec.



Select **No** for **Is Physical Device Available** and click **OK**.


Add Test Device

Is Physical Device Available

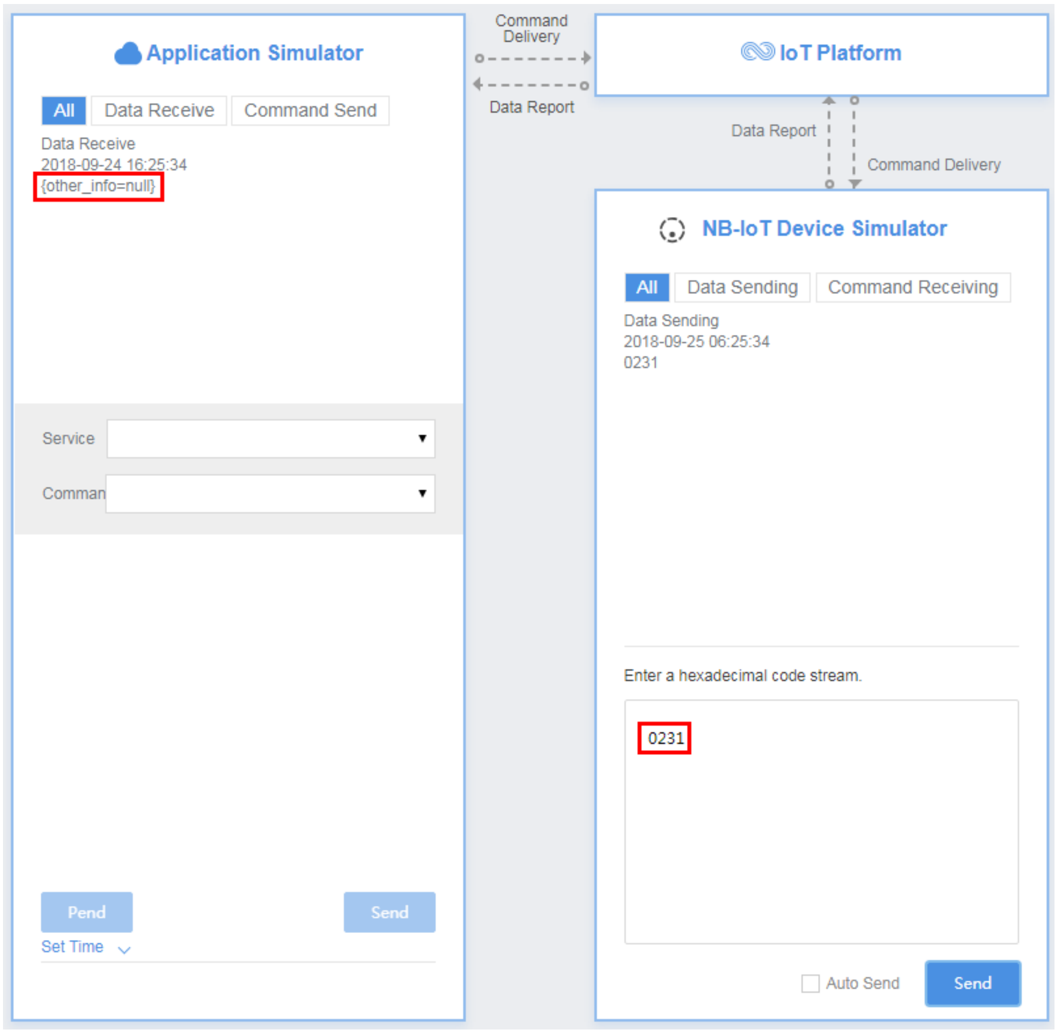
Yes [✓ No]

You are registering a virtual device.

Step 2 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.



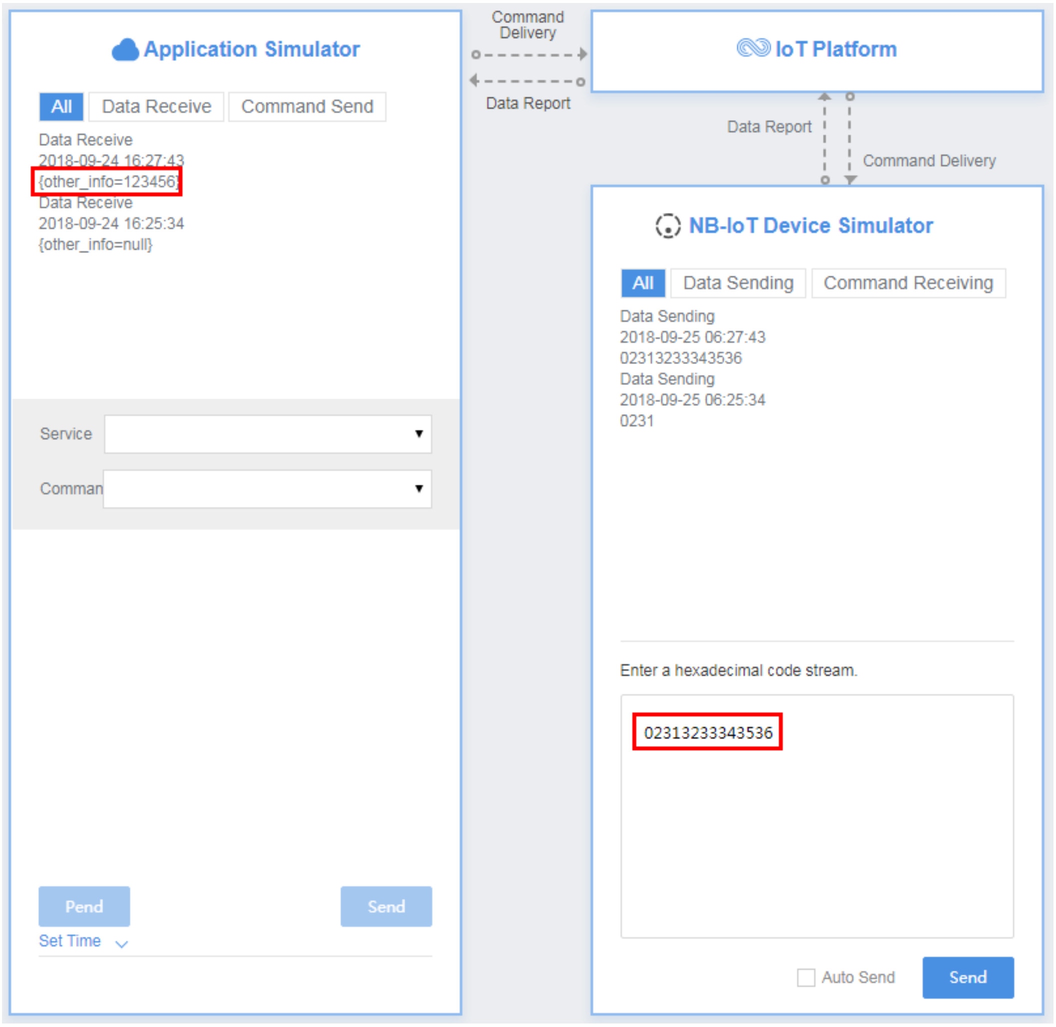
Data Receive
2018-09-24 16:25:34
{other_info=null}

Data Sending
2018-09-25 06:25:34
0231

0231

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.



```
Data Receive
2018-09-24 16:27:43
{other_info=123456}
Data Receive
2018-09-24 16:25:34
{other_info=null}
```

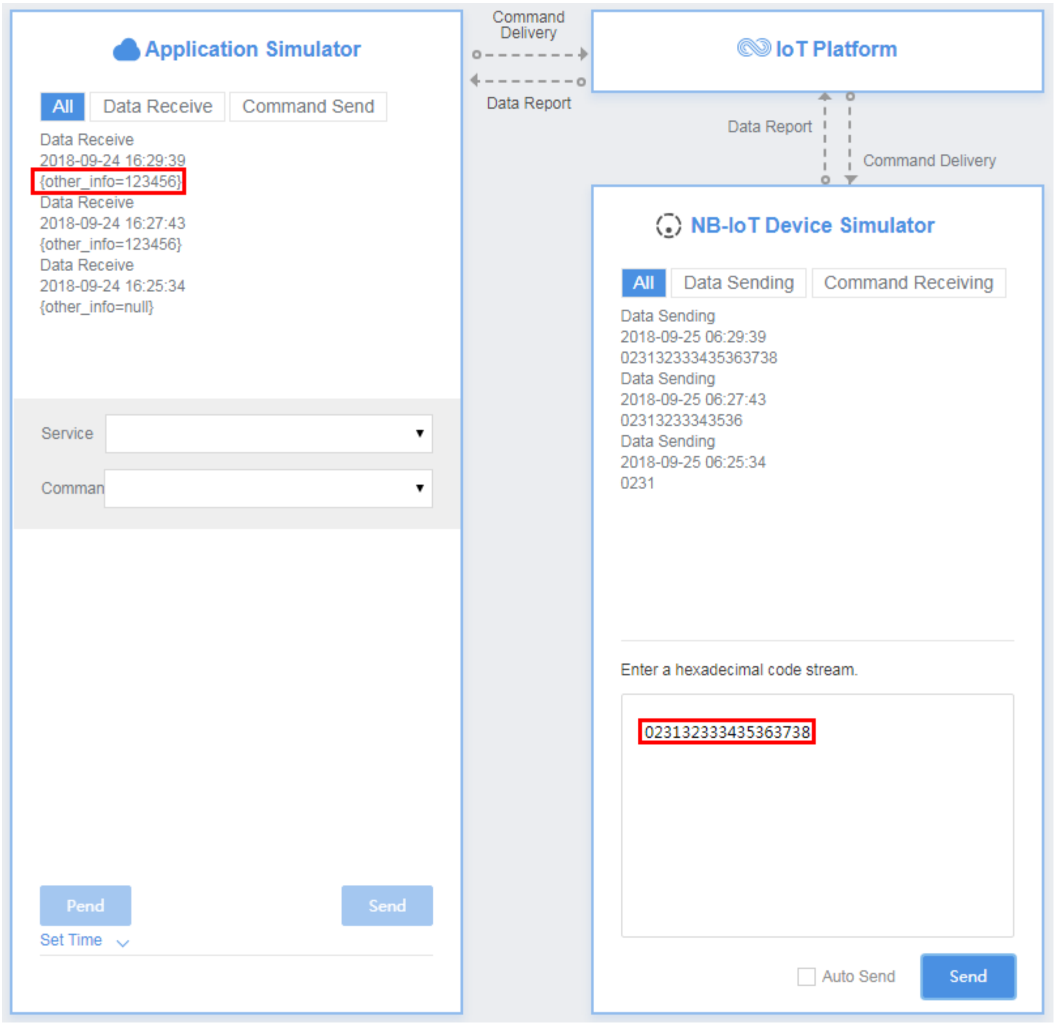
```
Data Sending
2018-09-25 06:27:43
02313233343536
Data Sending
2018-09-25 06:25:34
0231
```

Enter a hexadecimal code stream.

```
02313233343536
```

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.



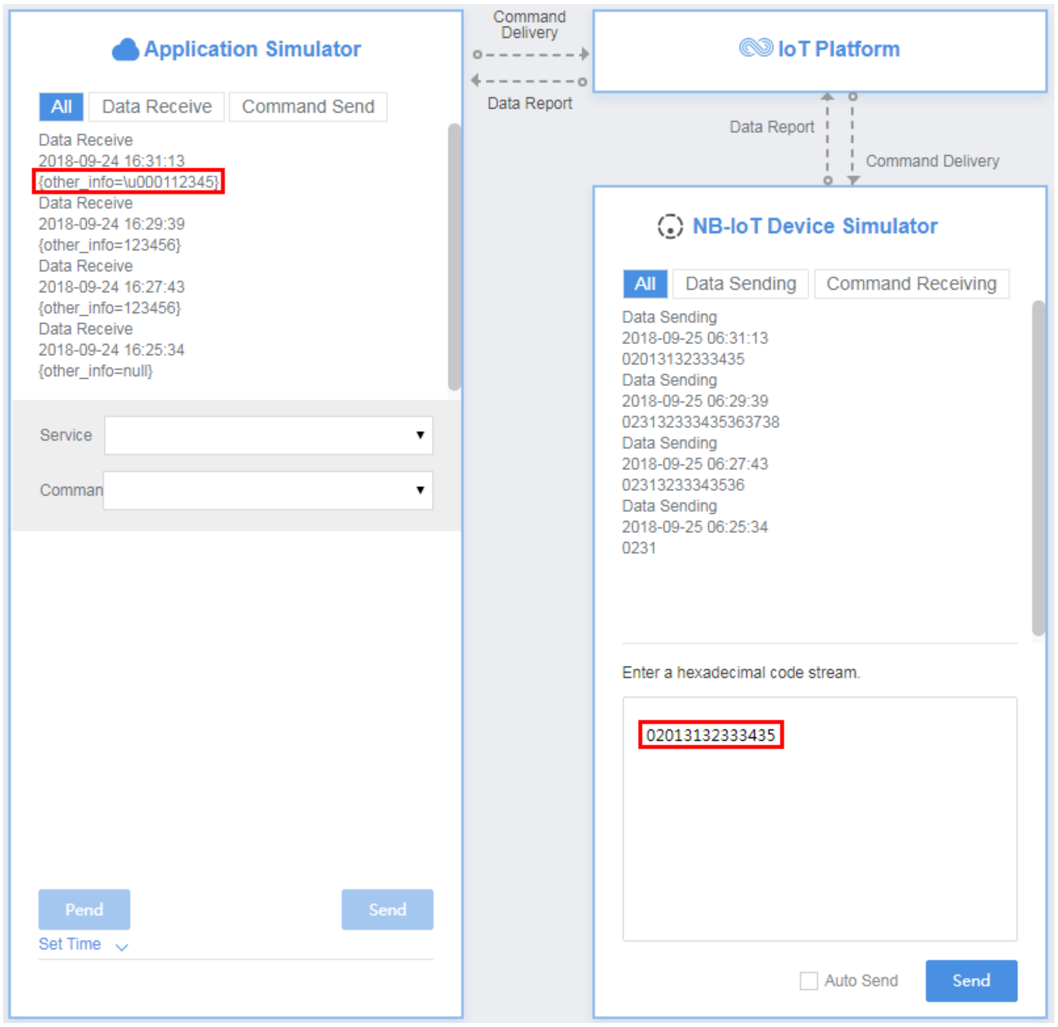
```
Data Receive
2018-09-24 16:29:39
{other_info=123456}
Data Receive
2018-09-24 16:27:43
{other_info=123456}
Data Receive
2018-09-24 16:25:34
{other_info=null}
```

```
Data Sending
2018-09-25 06:29:39
0231323333435363738
Data Sending
2018-09-25 06:27:43
02313233343536
Data Sending
2018-09-25 06:25:34
0231
```

```
0231323333435363738
```

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.



```
Data Receive
2018-09-24 16:31:13
{other_info=\u000112345}
Data Receive
2018-09-24 16:29:39
{other_info=123456}
Data Receive
2018-09-24 16:27:43
{other_info=123456}
Data Receive
2018-09-24 16:25:34
{other_info=null}
```

```
Data Sending
2018-09-25 06:31:13
02013132333435
Data Sending
2018-09-25 06:29:39
0231323334353637 38
Data Sending
2018-09-25 06:27:43
02313233343536
Data Sending
2018-09-25 06:25:34
0231
```

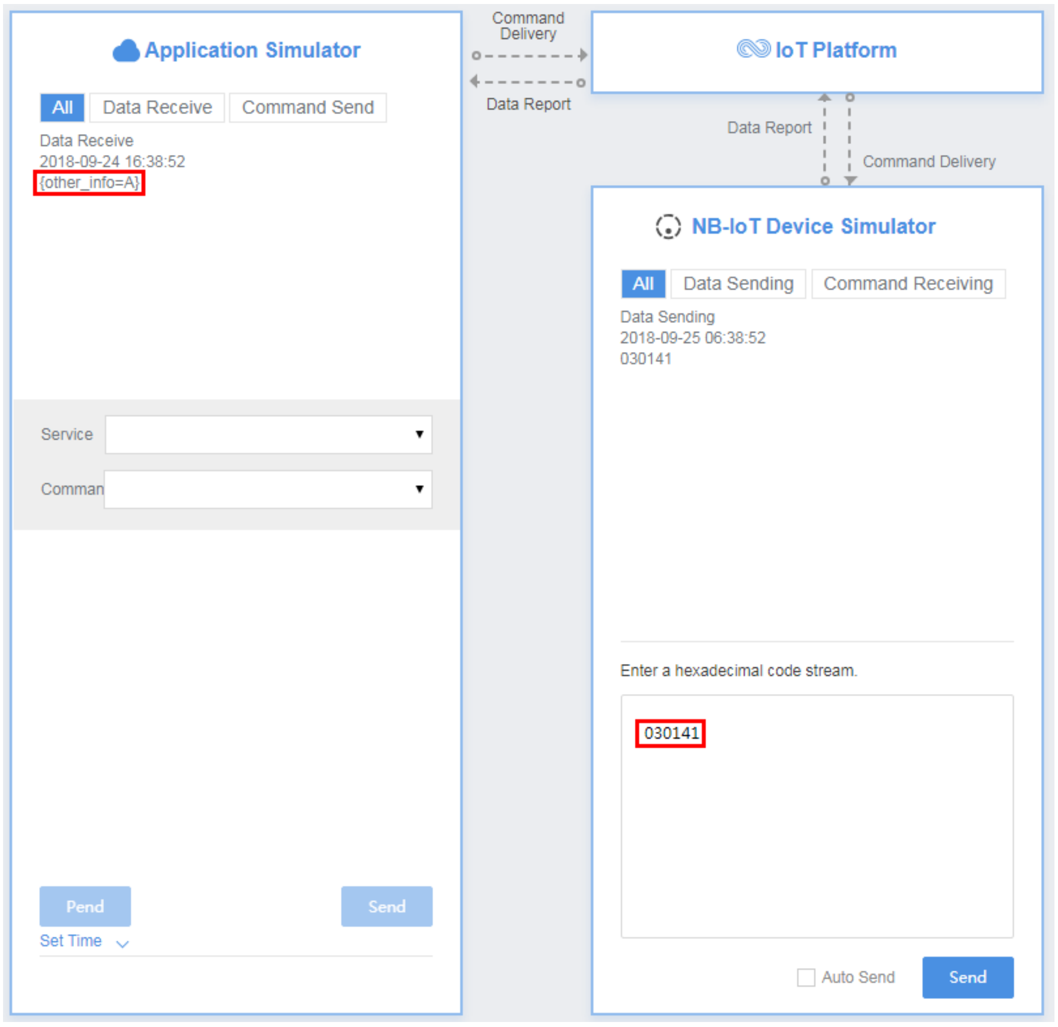
Enter a hexadecimal code stream.

```
02013132333435
```

Step 3 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.



Data Receive
2018-09-24 16:38:52
{other_info=A}

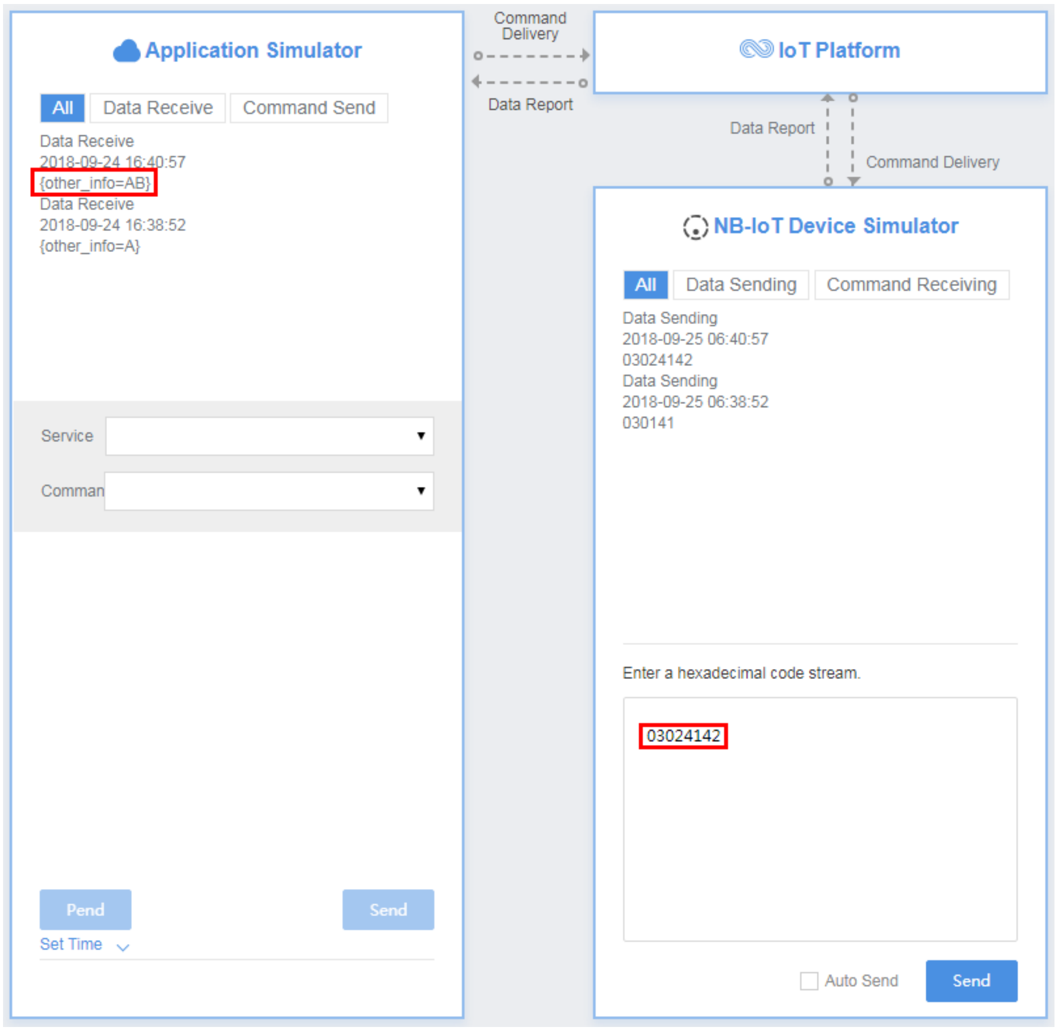
Data Sending
2018-09-25 06:38:52
030141

Enter a hexadecimal code stream.

030141

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.



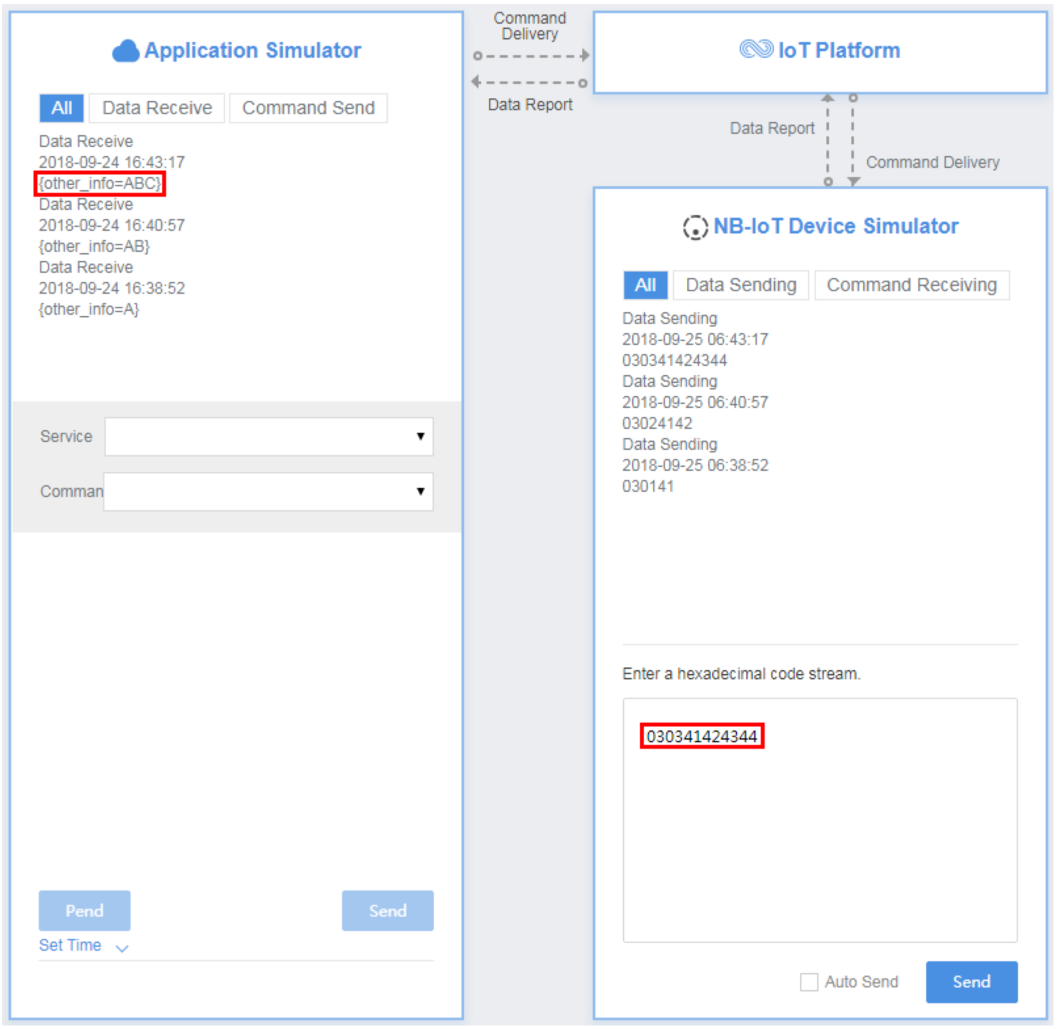
Data Receive
2018-09-24 16:40:57
{other_info=AB}
Data Receive
2018-09-24 16:38:52
{other_info=A}

Data Sending
2018-09-25 06:40:57
03024142
Data Sending
2018-09-25 06:38:52
030141

03024142

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.



```
Data Receive
2018-09-24 16:43:17
{other_info=ABC}
Data Receive
2018-09-24 16:40:57
{other_info=AB}
Data Receive
2018-09-24 16:38:52
{other_info=A}
```

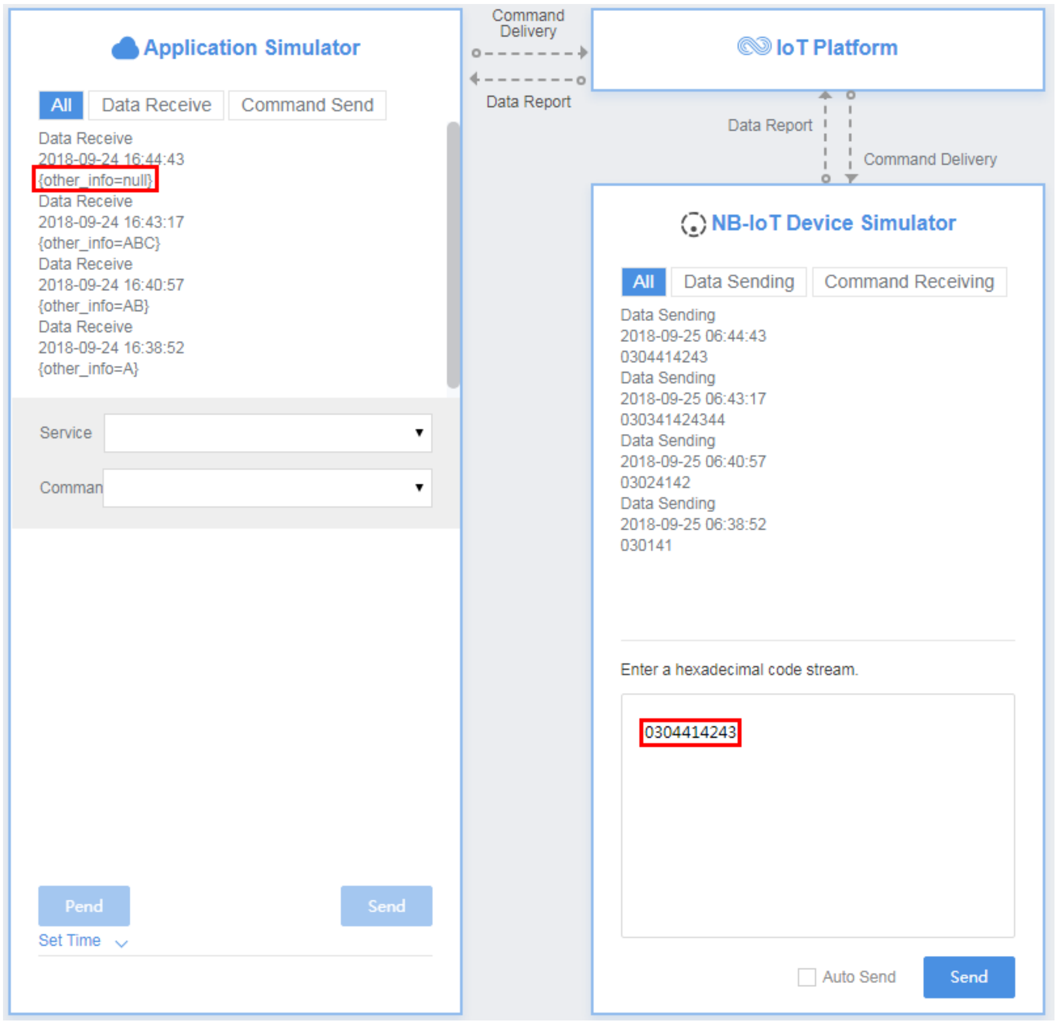
```
Data Sending
2018-09-25 06:43:17
030341424344
Data Sending
2018-09-25 06:40:57
03024142
Data Sending
2018-09-25 06:38:52
030141
```

Enter a hexadecimal code stream.

030341424344

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.



```
Data Receive
2018-09-24 16:44:43
{other_info=null}
Data Receive
2018-09-24 16:43:17
{other_info=ABC}
Data Receive
2018-09-24 16:40:57
{other_info=AB}
Data Receive
2018-09-24 16:38:52
{other_info=A}
```

```
Data Sending
2018-09-25 06:44:43
0304414243
Data Sending
2018-09-25 06:43:17
030341424344
Data Sending
2018-09-25 06:40:57
03024142
Data Sending
2018-09-25 06:38:52
030141
```

```
0304414243
```

----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.
- Online developed codecs 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.

1.4.3.4 Codec for Arrays and Variable-Length Arrays

Scenarios

A smoke detector provides the following functions:

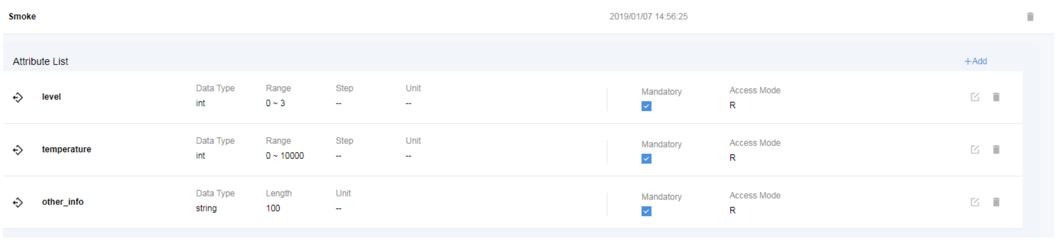
- Reporting smoke alarms (fire severity) and temperature simultaneously, or reporting the temperature separately.
- Reporting description. The data type of description can be **array (array type)** or **variant (variable-length array type)**.

  NOTE

  This scenario describes how to develop a codec for data in arrays and data in variable-length arrays. The data reporting and command delivery codecs are developed in the same way. Therefore, data reporting is used as an example and command delivery is not described.

Defining the Profile File

Define the profile file in the development space of the smoke sensor.


Smoke 2019/01/07 14:56:25

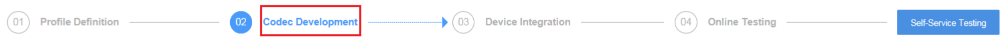
Attribute List +Add

level	Data Type: int	Range: 0 ~ 3	Step: --	Unit: --	Mandatory: ✓	Access Mode: R
temperature	Data Type: int	Range: 0 ~ 10000	Step: --	Unit: --	Mandatory: ✓	Access Mode: R
other_info	Data Type: string	Length: 100	Unit: --		Mandatory: ✓	Access Mode: R

Developing a Codec

This section describes only the procedure for developing the codec for reporting the description (**other_info**). For details on how to develop the codec for reporting the smoke alarms (**level**) and temperature (**temperature**), see **Codec for Multiple Data Reporting Messages**.

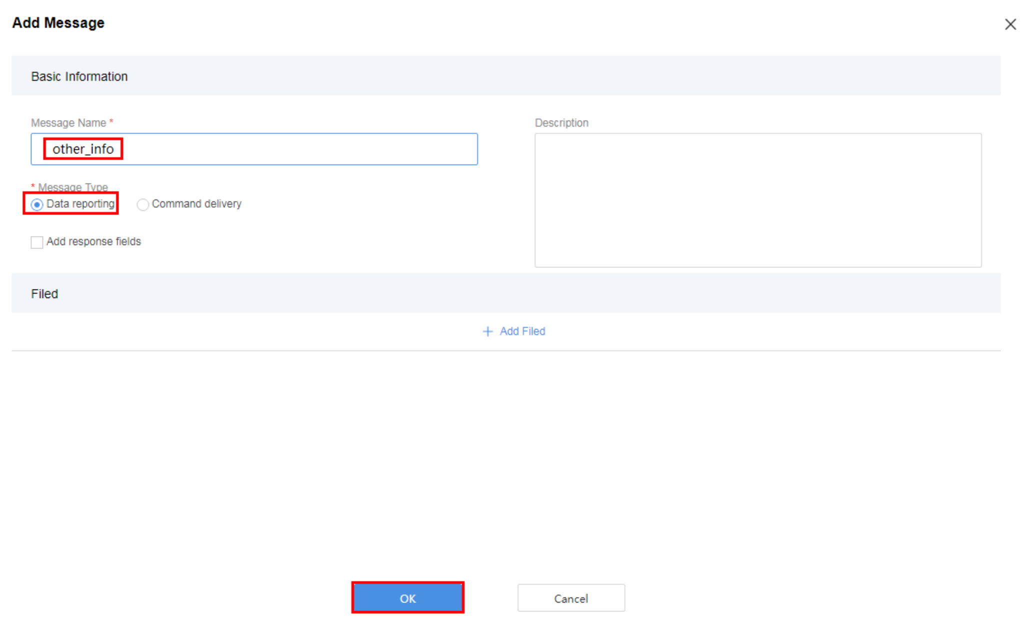
Step 1 In the development space of the smoke sensor, click **Codec Development**.



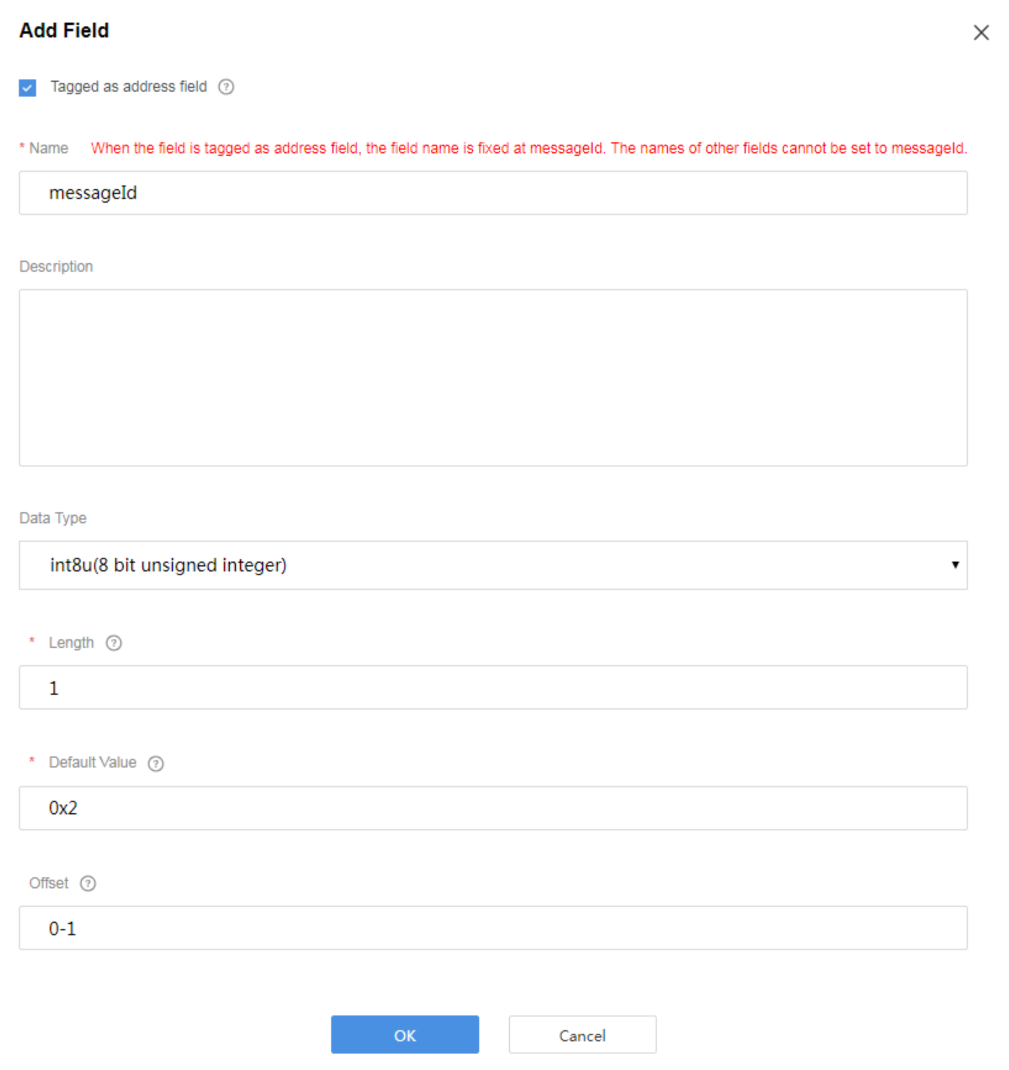
Step 2 Configure a data reporting message to report the fire severity and temperature. For details, see **Step 2**.

Step 3 Configure a data reporting message to report only the temperature. For details, see **Step 3**.

Step 4 Configure a data reporting message to report the description of the array type.



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 **0x2** is used to identify the message that reports the description (of the array type).



☑ Tagged as address field

* Name When the field is tagged as address field, the field name is fixed at messageId. The names of other fields cannot be set to messageId.

messageId

Description

Data Type

int8u(8 bit unsigned integer)

* Length

1

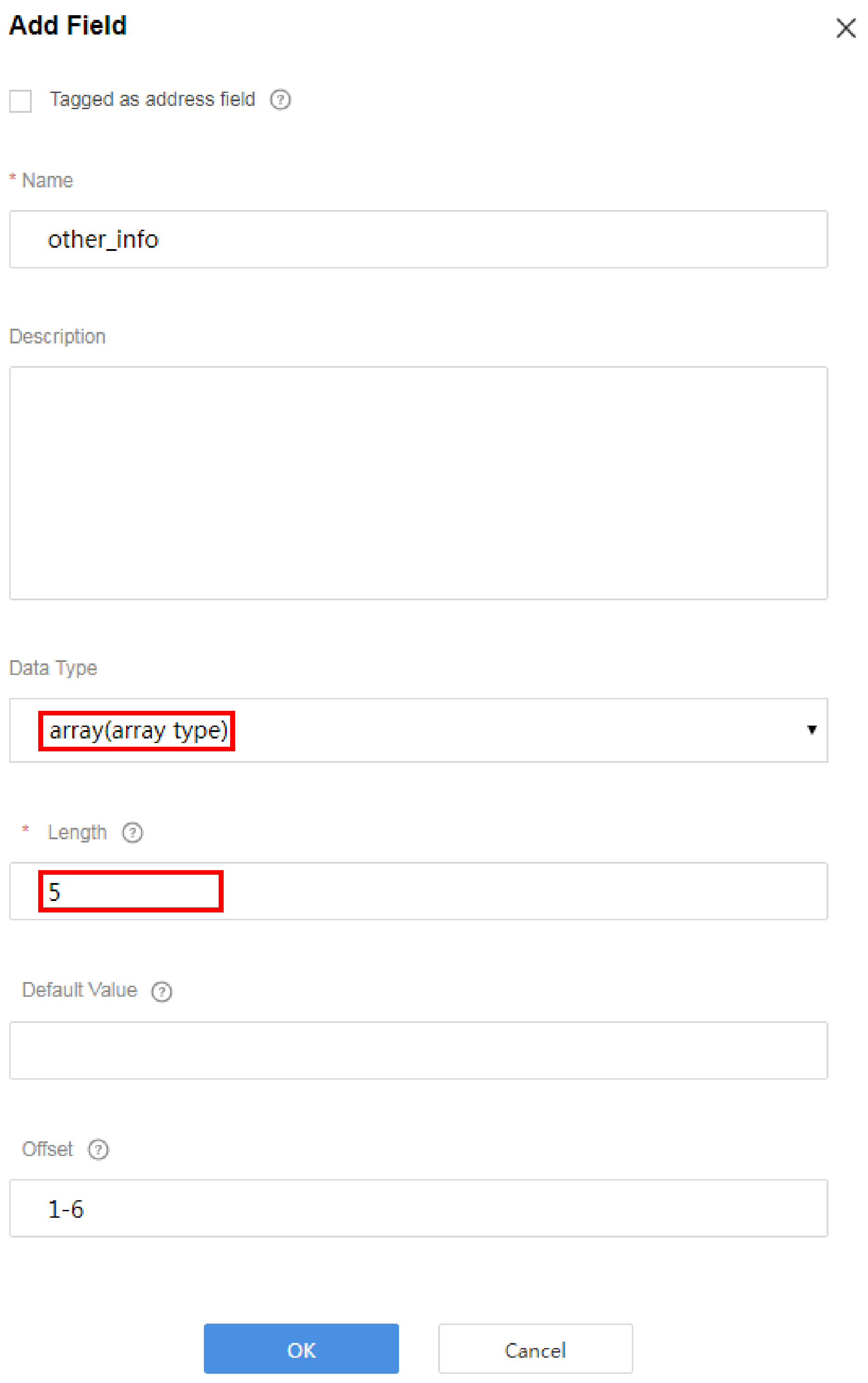
* Default Value

0x2

Offset

0-1

Add the **other_info** field to indicate the description of the array type. In this scenario, set **Length** to **5**.


Add Field

Tagged as address field

* Name

other_info

Description

Data Type

array(array type)

* Length

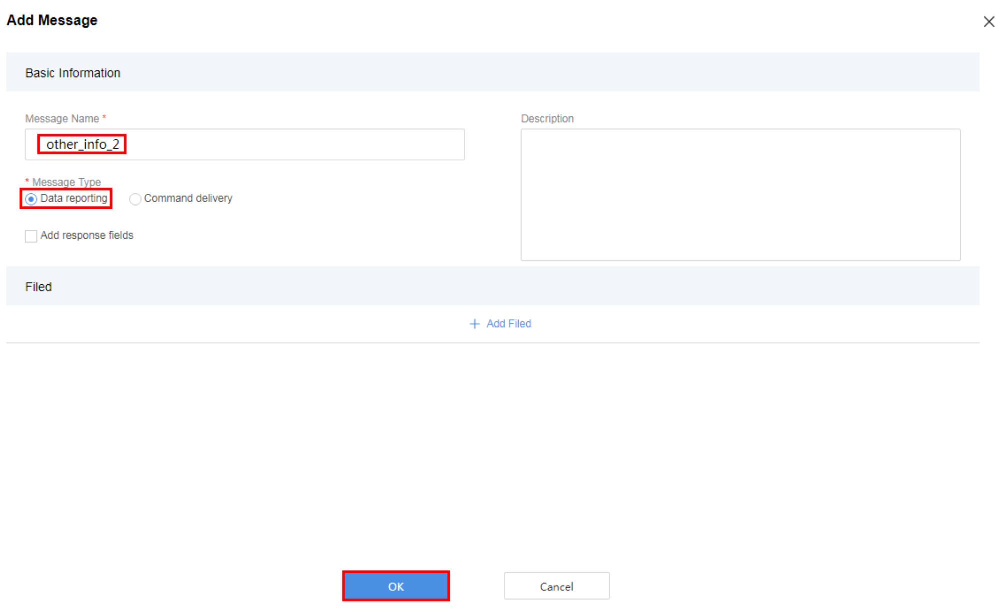
5

Default Value

Offset

1-6

Step 5 Configure a data reporting message to report the description of the variable-length array type.



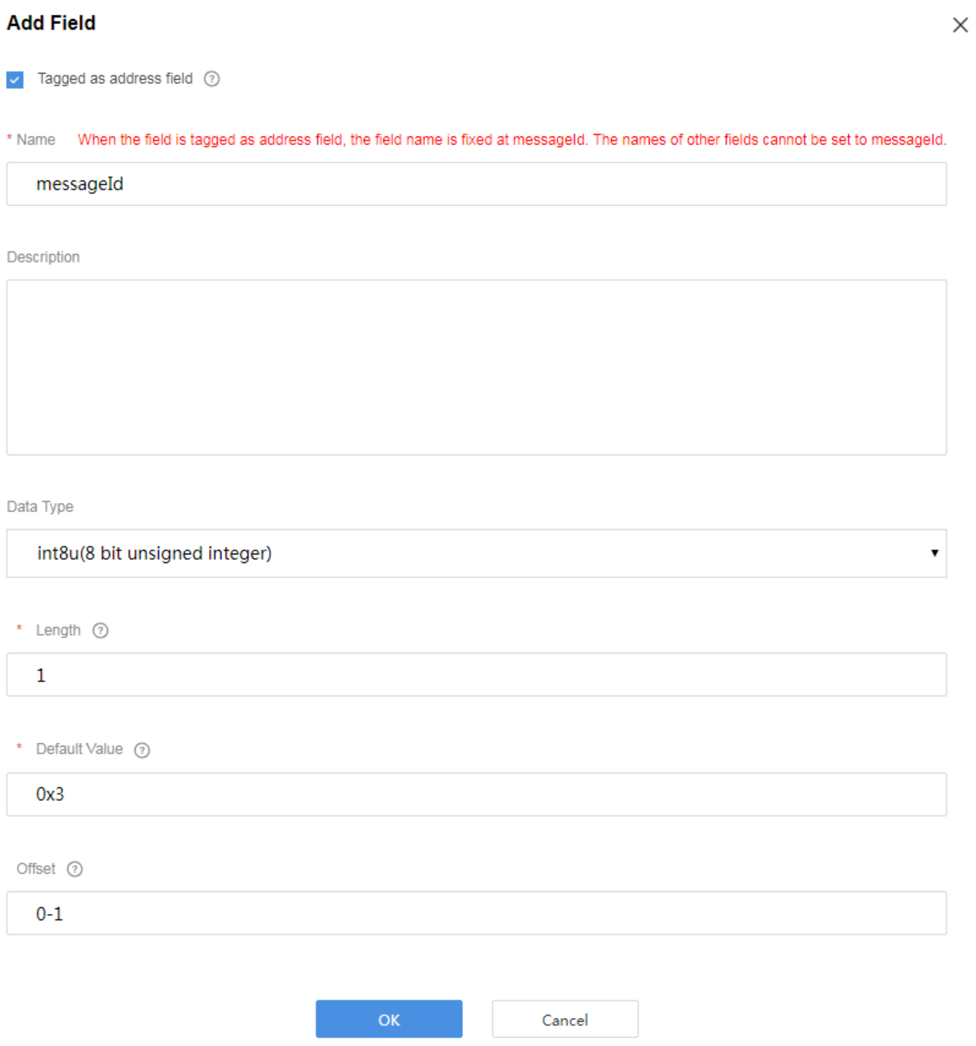
Message Name *: other_info_2

Description:

* Message Type: [✓ Data reporting] Command delivery

Add response fields

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 array type).


Add Field

✓ Tagged as address field

* Name When the field is tagged as address field, the field name is fixed at messageId. The names of other fields cannot be set to messageId.

messageId

Description

Data Type

int8u(8 bit unsigned integer)

* Length

1

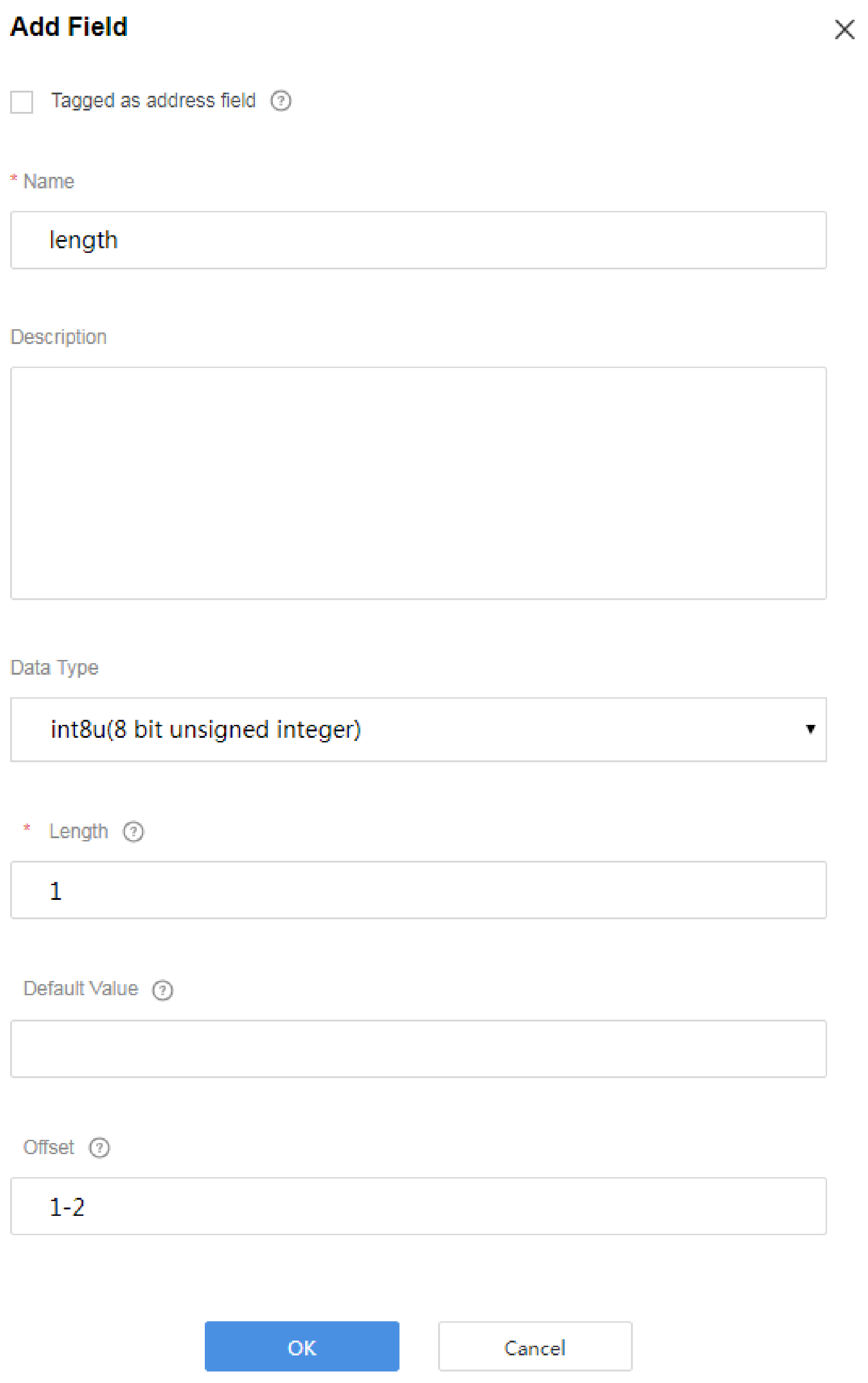
* Default Value

0x3

Offset

0-1

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**.


Add Field

☐ Tagged as address field

* Name: length

Description:

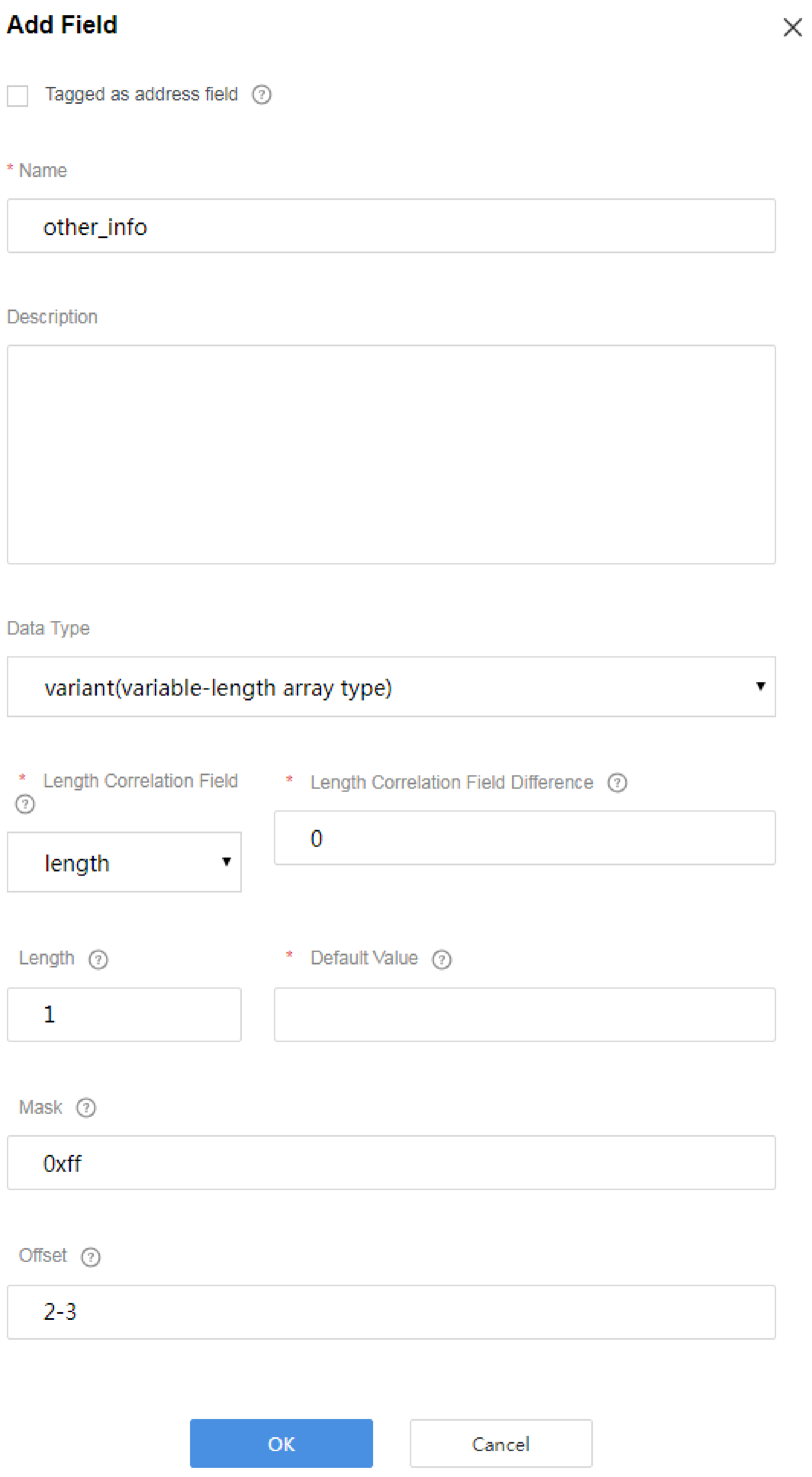
Data Type: int8u(8 bit unsigned integer)

* Length: 1

Default Value:

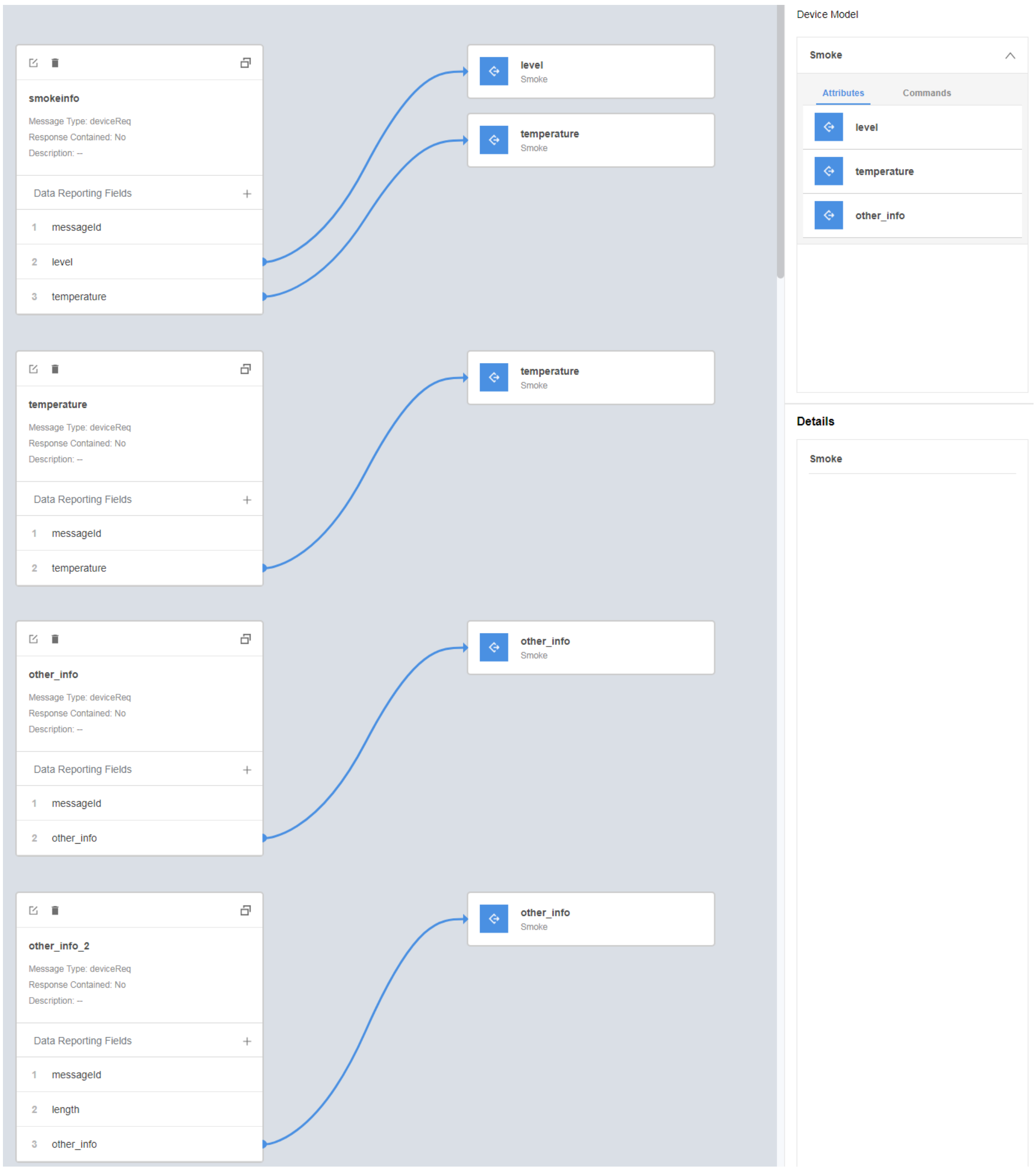
Offset: 1-2

Add the **other_info** field to indicate 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.

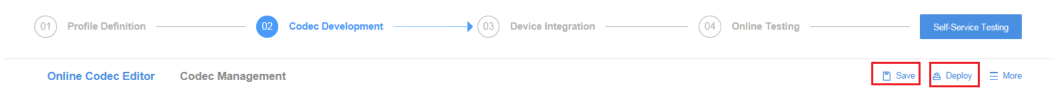


Tagged as address field	☐
* Name	other_info
Description	
Data Type	variant(variable-length array type)
* Length Correlation Field	length
* Length Correlation Field Difference	0
Length	1
* Default Value	
Mask	0xff
Offset	2-3

Step 6 Drag the property fields in **Device Model** on the right to set up a mapping with the corresponding fields in the data reporting messages.



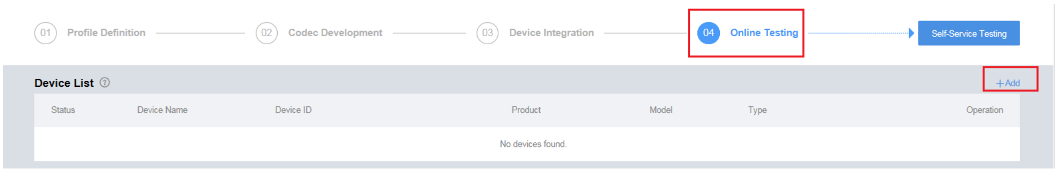
Step 7 Click **Save** and then **Deploy** to deploy the codec on the IoT platform.



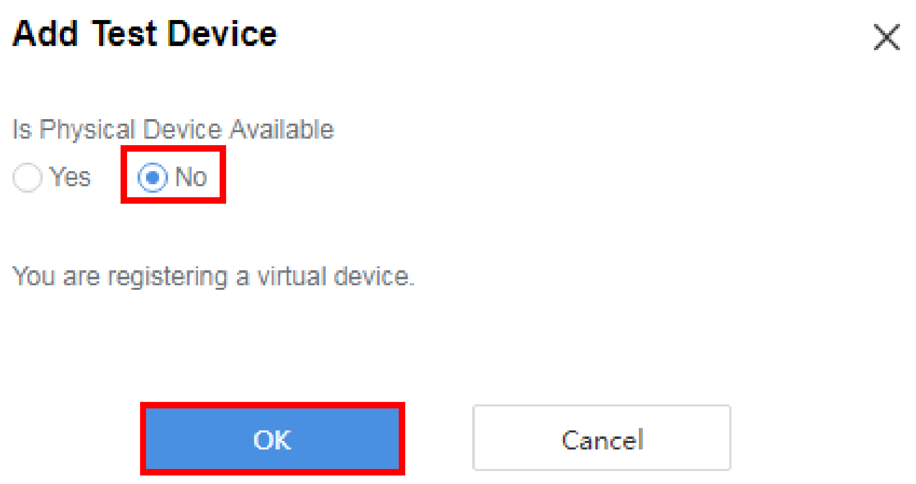
----End

Testing the Codec

Step 1 In the development space of the smoke sensor, click **Online Testing** and add a virtual device to test the codec.



Select **No** for **Is Physical Device Available** and click **OK**.



Add Test Device

Is Physical Device Available

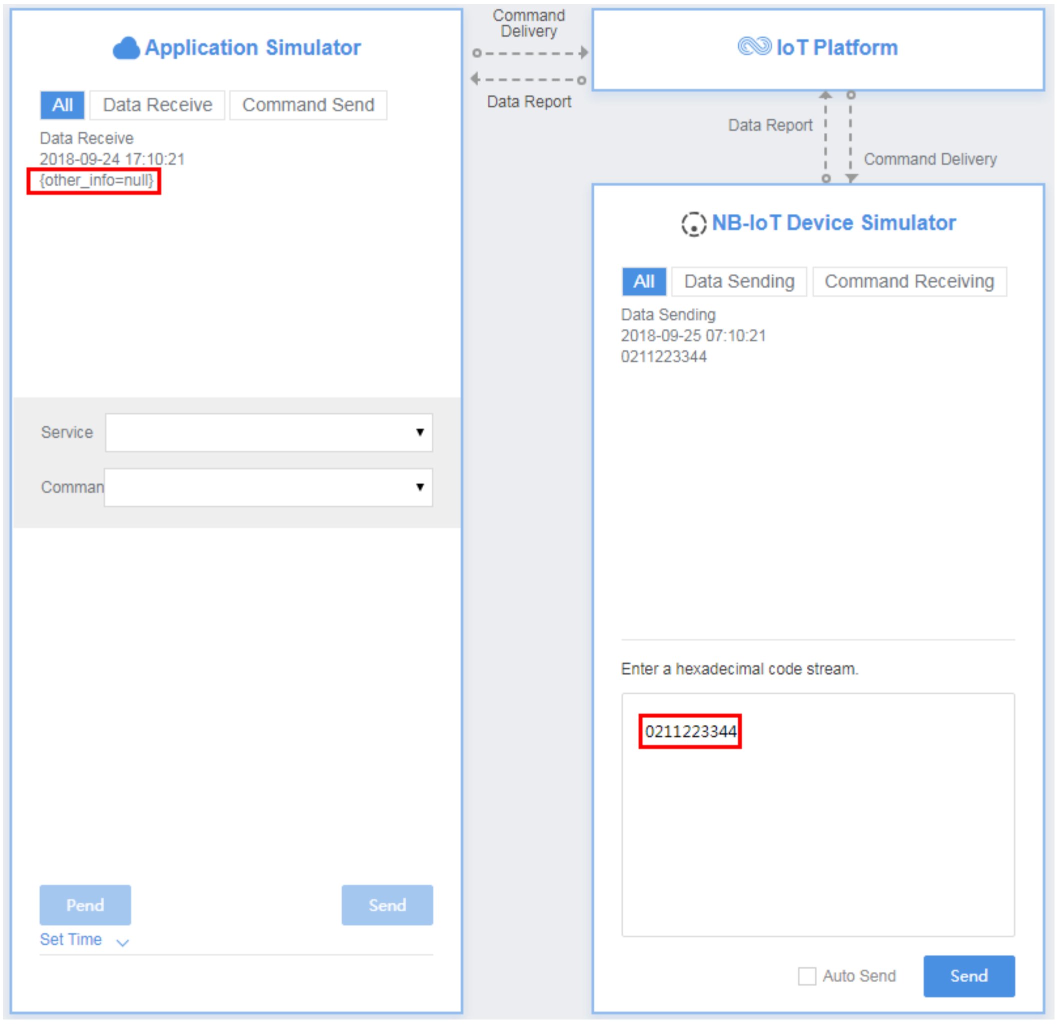
Yes [✓ No]

You are registering a virtual device.

Step 2 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.



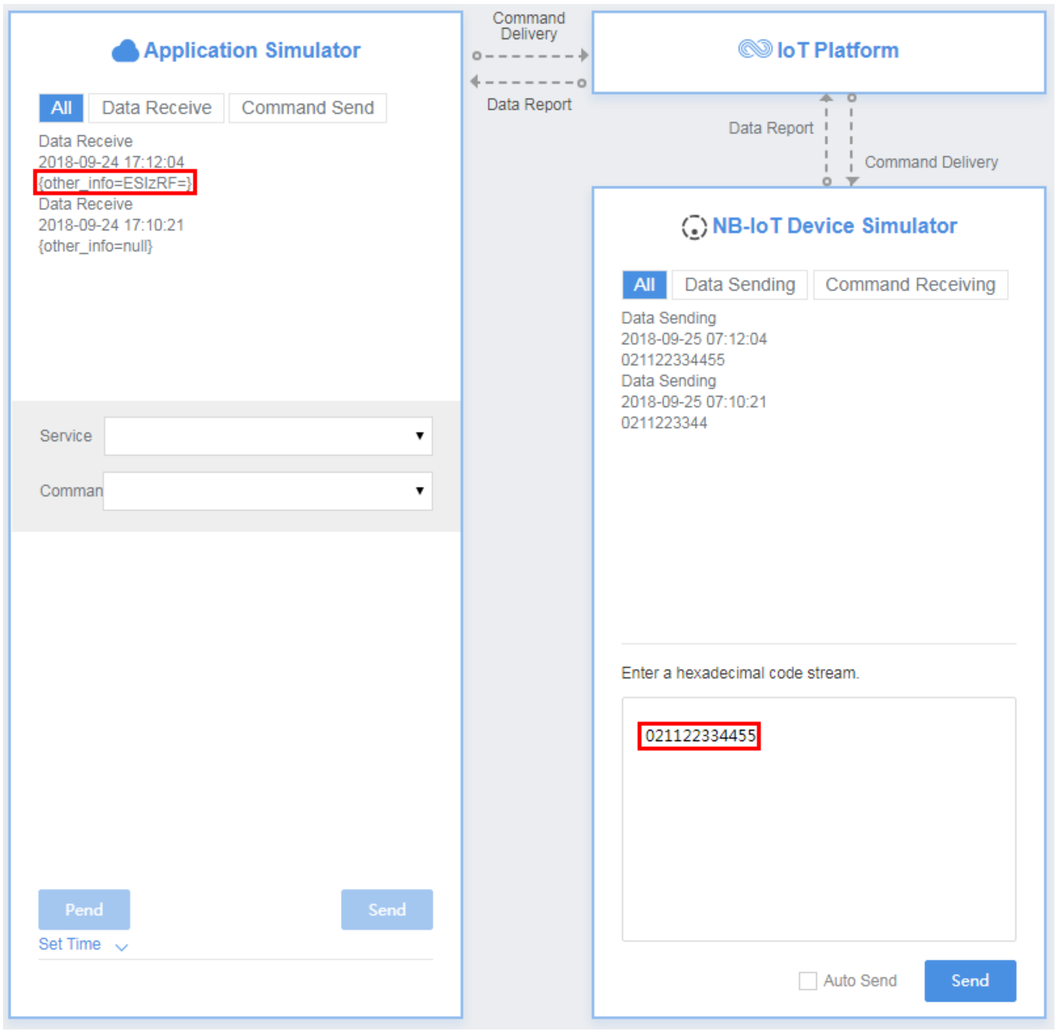
Data Receive
2018-09-24 17:10:21
{other_info=null}

Data Sending
2018-09-25 07:10:21
0211223344

0211223344

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.



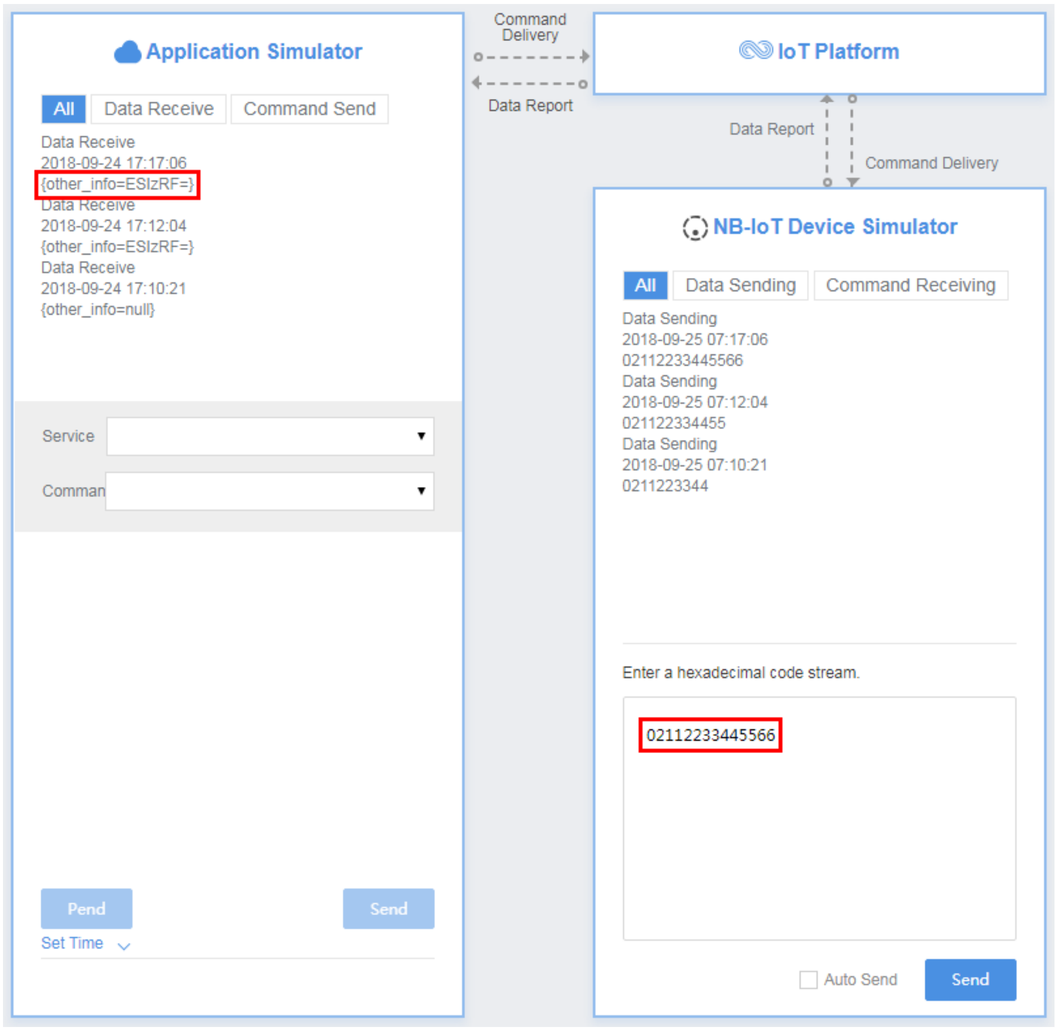
Data Receive
2018-09-24 17:12:04
{other_info=ESIzRF=}
Data Receive
2018-09-24 17:10:21
{other_info=null}

Data Sending
2018-09-25 07:12:04
021122334455
Data Sending
2018-09-25 07:10:21
0211223344

021122334455

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.



```
Data Receive
2018-09-24 17:17:06
{other_info=ESIzRF=}
Data Receive
2018-09-24 17:12:04
{other_info=ESIzRF=}
Data Receive
2018-09-24 17:10:21
{other_info=null}
```

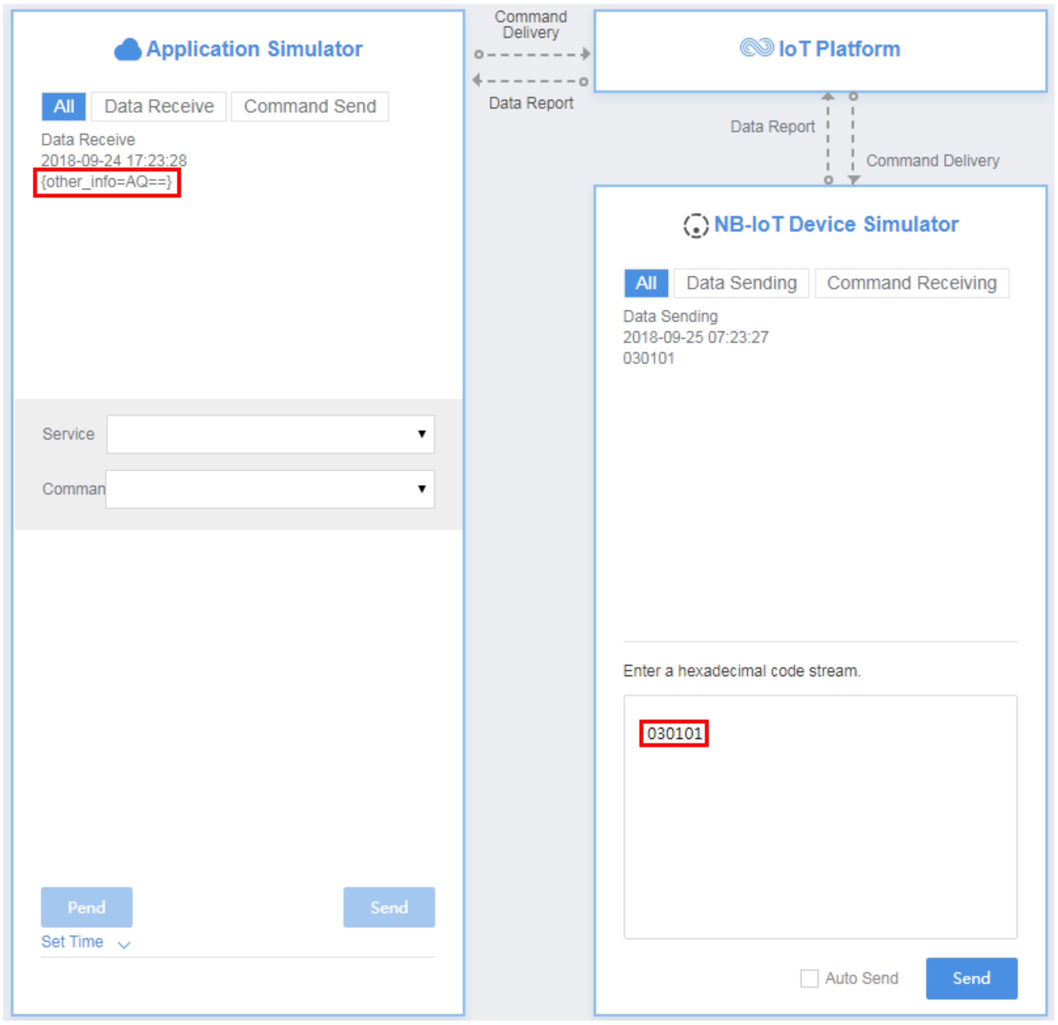
```
Data Sending
2018-09-25 07:17:06
02112233445566
Data Sending
2018-09-25 07:12:04
021122334455
Data Sending
2018-09-25 07:10:21
0211223344
```

```
02112233445566
```

Step 3 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.



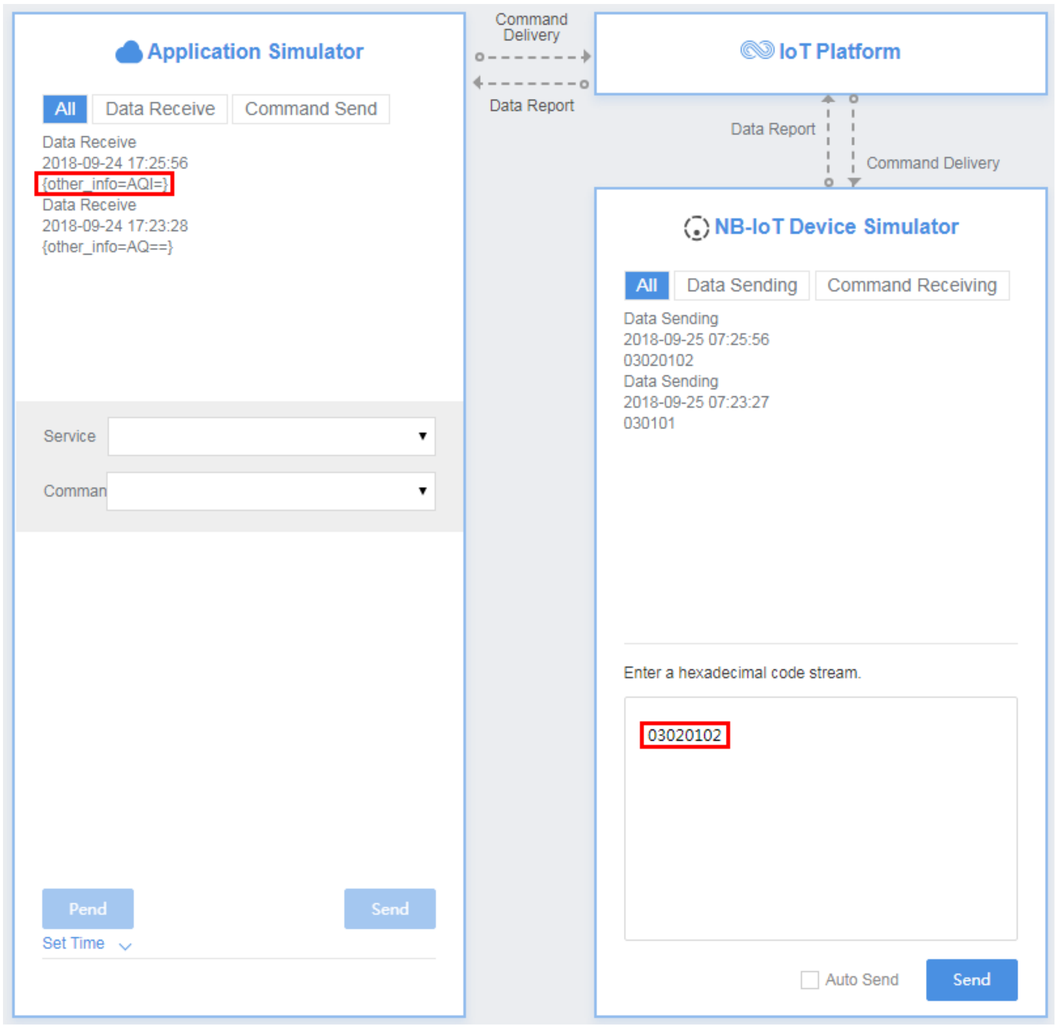
Data Receive
2018-09-24 17:23:28
{other_info=AQ==}

Data Sending
2018-09-25 07:23:27
030101

030101

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.



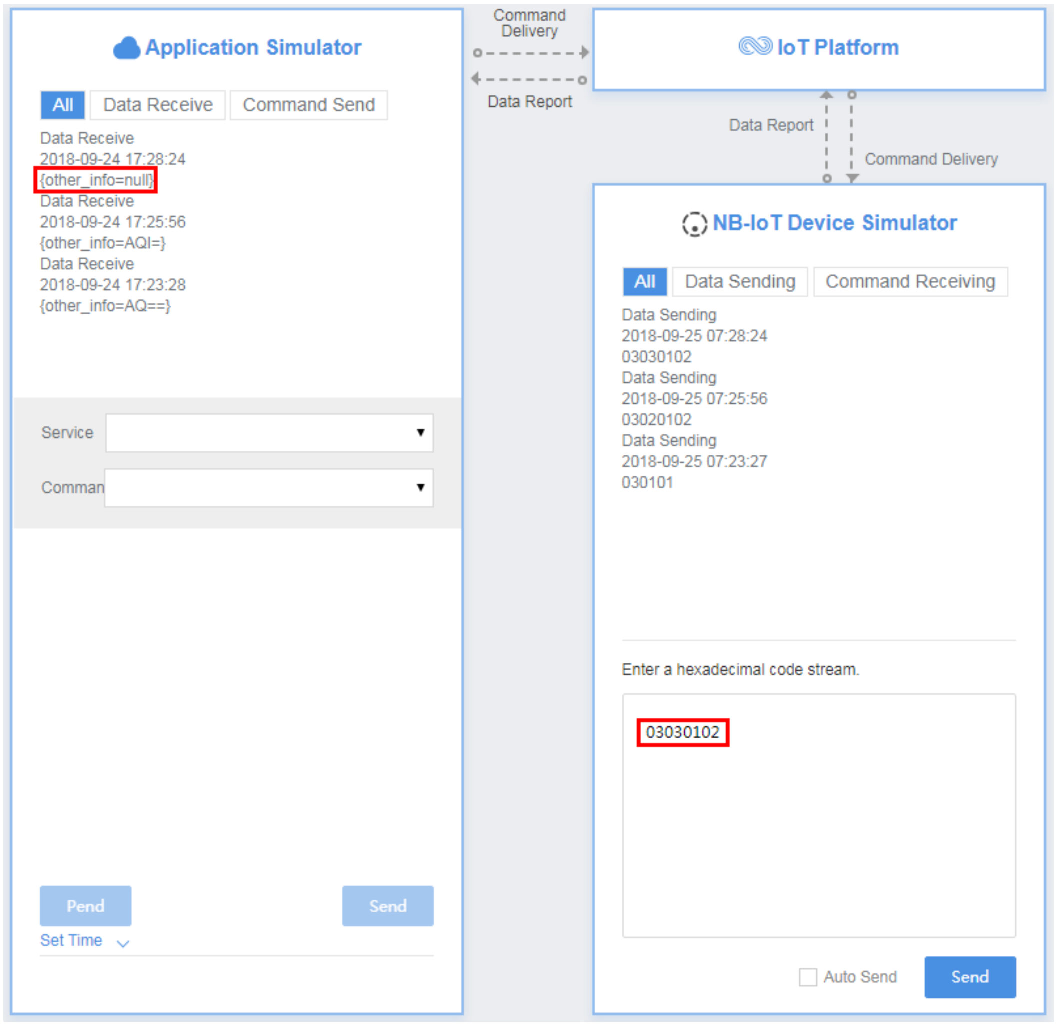
Data Receive
2018-09-24 17:25:56
{other_info=AQI=}
Data Receive
2018-09-24 17:23:28
{other_info=AQ==}

Data Sending
2018-09-25 07:25:56
03020102
Data Sending
2018-09-25 07:23:27
030101

03020102

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.



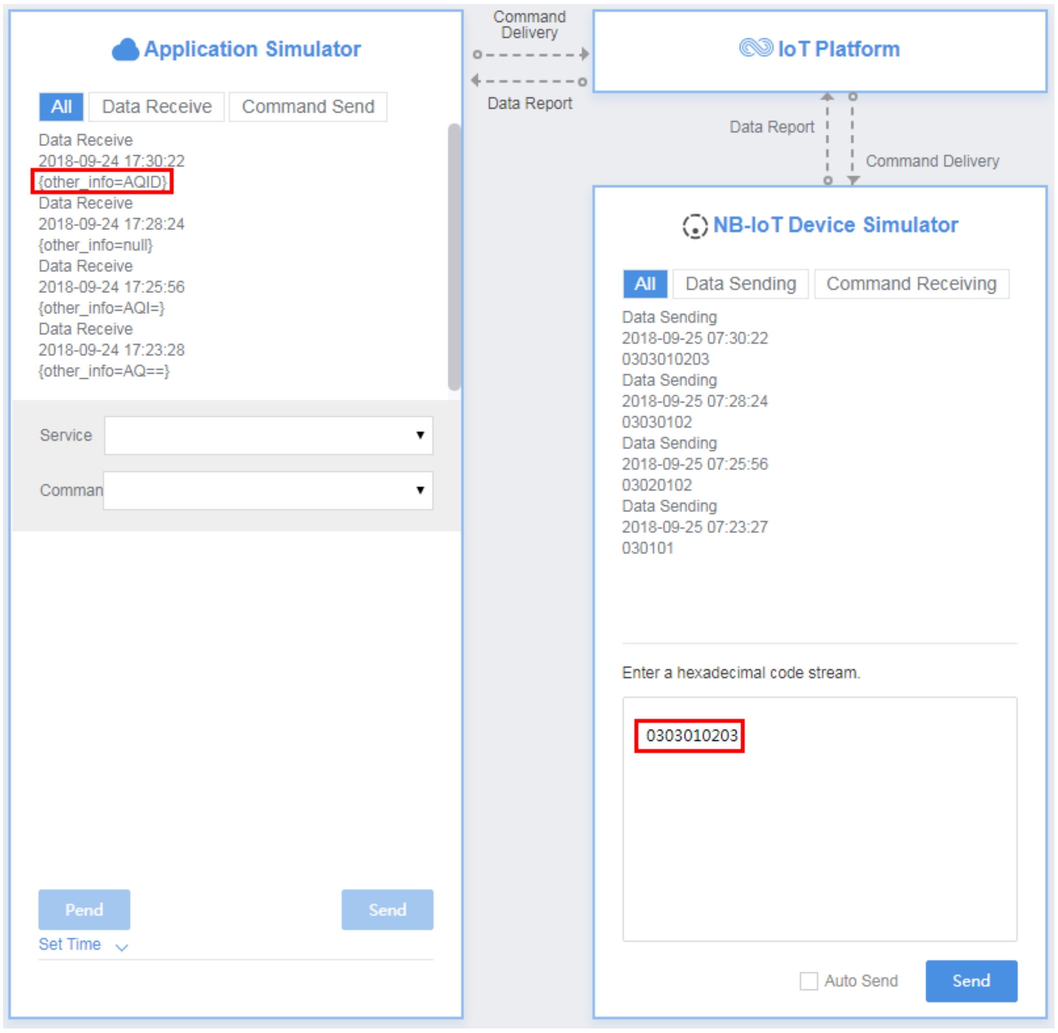
```
Data Receive
2018-09-24 17:28:24
{other_info=null}
Data Receive
2018-09-24 17:25:56
{other_info=AQI=}
Data Receive
2018-09-24 17:23:28
{other_info=AQ==}
```

```
Data Sending
2018-09-25 07:28:24
03030102
Data Sending
2018-09-25 07:25:56
03020102
Data Sending
2018-09-25 07:23:27
030101
```

```
03030102
```

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.



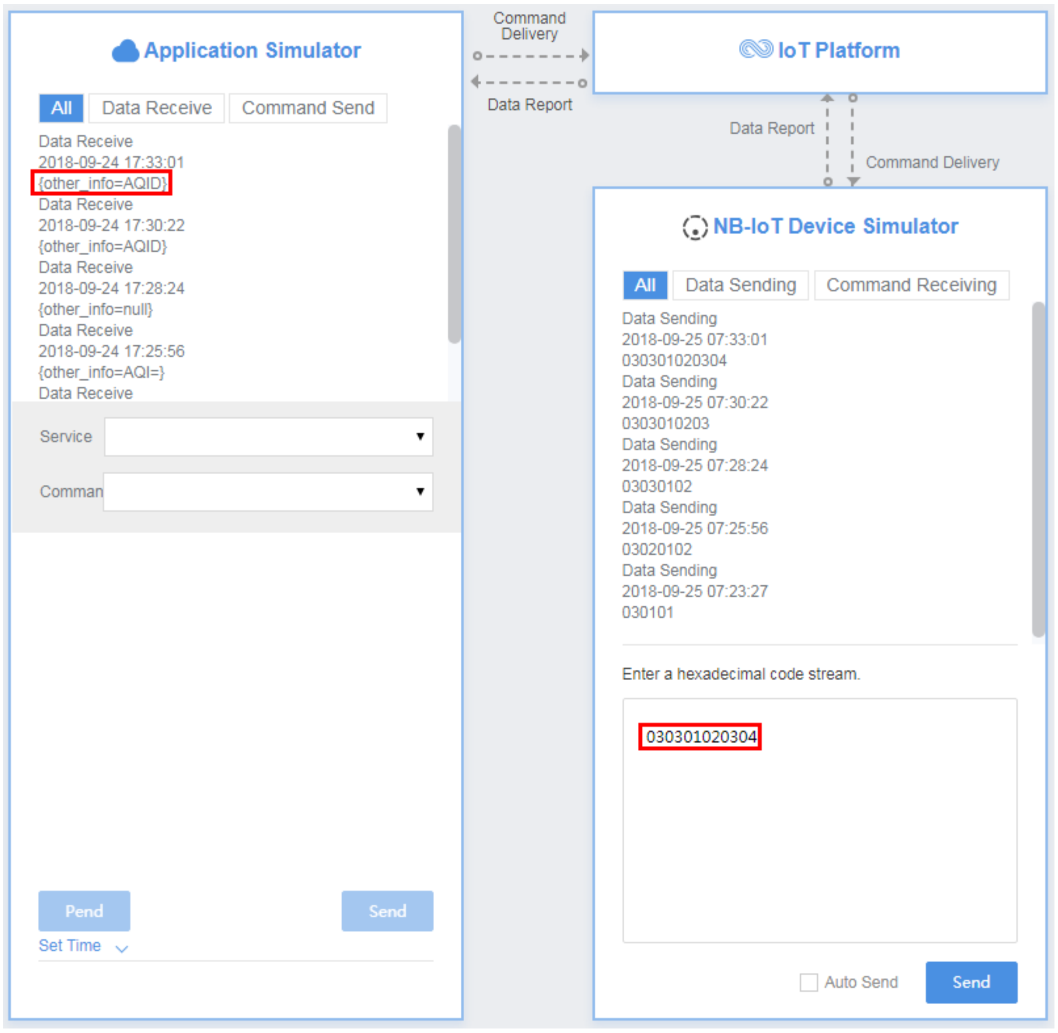
```
Data Receive
2018-09-24 17:30:22
{other_info=AQID}
Data Receive
2018-09-24 17:28:24
{other_info=null}
Data Receive
2018-09-24 17:25:56
{other_info=AQI=}
Data Receive
2018-09-24 17:23:28
{other_info=AQ==}
```

```
Data Sending
2018-09-25 07:30:22
0303010203
Data Sending
2018-09-25 07:28:24
03030102
Data Sending
2018-09-25 07:25:56
03020102
Data Sending
2018-09-25 07:23:27
030101
```

```
0303010203
```

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.



```
Data Receive
2018-09-24 17:33:01
{other_info=AQID}
Data Receive
2018-09-24 17:30:22
{other_info=AQID}
Data Receive
2018-09-24 17:28:24
{other_info=null}
Data Receive
2018-09-24 17:25:56
{other_info=AQI=}
Data Receive
```

```
Data Sending
2018-09-25 07:33:01
030301020304
Data Sending
2018-09-25 07:30:22
0303010203
Data Sending
2018-09-25 07:28:24
03030102
Data Sending
2018-09-25 07:25:56
03020102
Data Sending
2018-09-25 07:23:27
030101
```

030301020304

----End

Description of Base64 Encoding Modes

In the Base64 encoding modes, 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. 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 characters. It 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 mark (=). 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). Then, convert the binary number into four 6-bit numbers: 000000, 010000, 000000, and 000000. 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 marks (==). Finally, **AQ==** is obtained by using **01** as a value.

Summary

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

1.4.3.5 Codec for Containing Command Execution Results

Scenarios

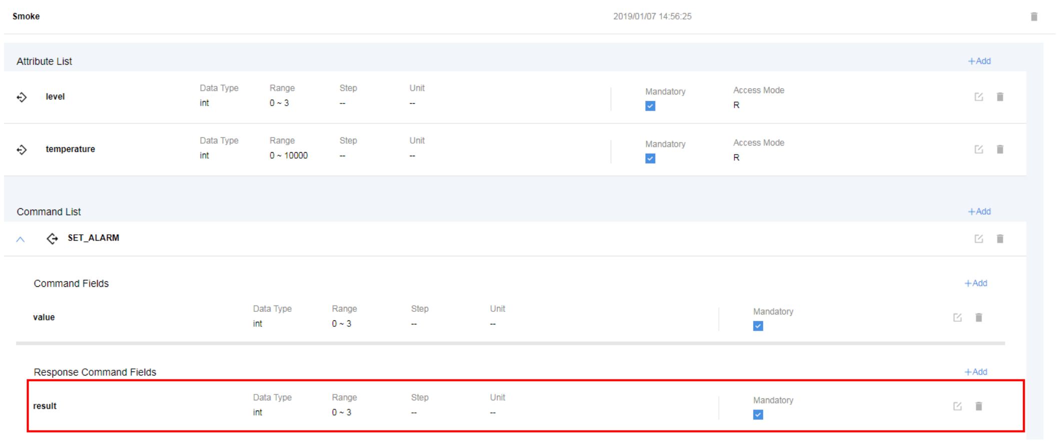
A smoke detector provides the following functions:

- Reporting smoke alarms (fire severity) and temperature
- Remote command, which can enable the alarm function remotely

  For example, the smoke detector can report the temperature on the fire scene and remotely trigger the smoke alarm for evacuation.
- Reporting command execution results

Defining the Profile File

Define the profile file in the development space of the smoke sensor.


Smoke 2019/01/07 14:56:25

Attribute List +Add

	Data Type	Range	Step	Unit	Mandatory	Access Mode
level	int	0 ~ 3	--	--	✓	R
temperature	int	0 ~ 10000	--	--	✓	R

Command List +Add

SET_ALARM

Command Fields +Add

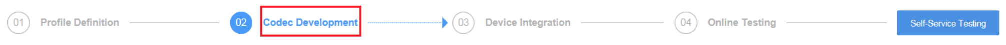
	Data Type	Range	Step	Unit	Mandatory
value	int	0 ~ 3	--	--	✓

Response Command Fields +Add

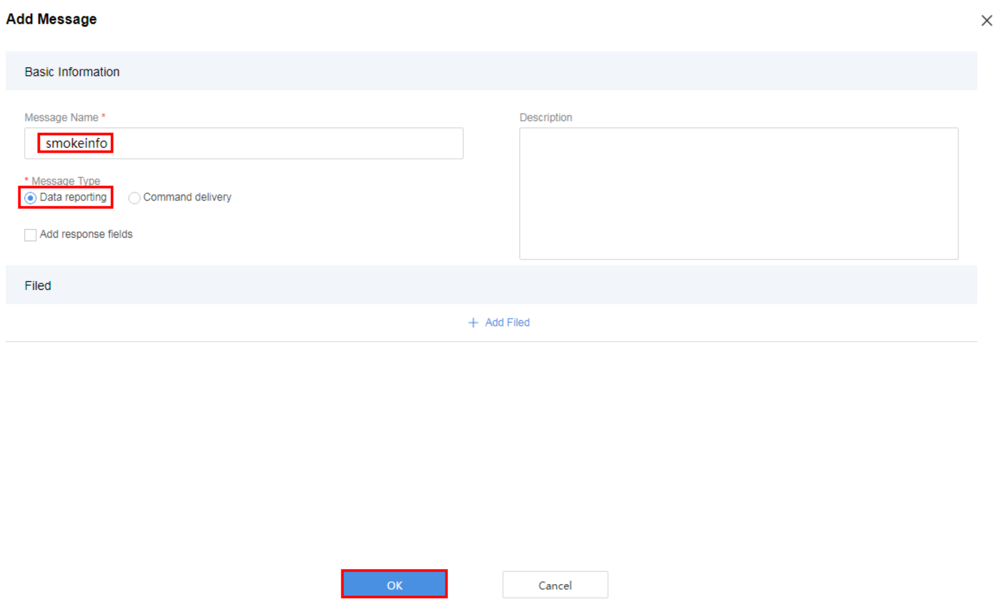
	Data Type	Range	Step	Unit	Mandatory
result	int	0 ~ 3	--	--	✓

Developing a Codec

Step 1 In the development space of the smoke sensor, click **Codec Development**.



Step 2 Configure a data reporting message to report the fire severity and temperature.


Add Message

Basic Information

Message Name * smokeinfo

Description

* Message Type [✓ Data reporting] Command delivery

Add response fields

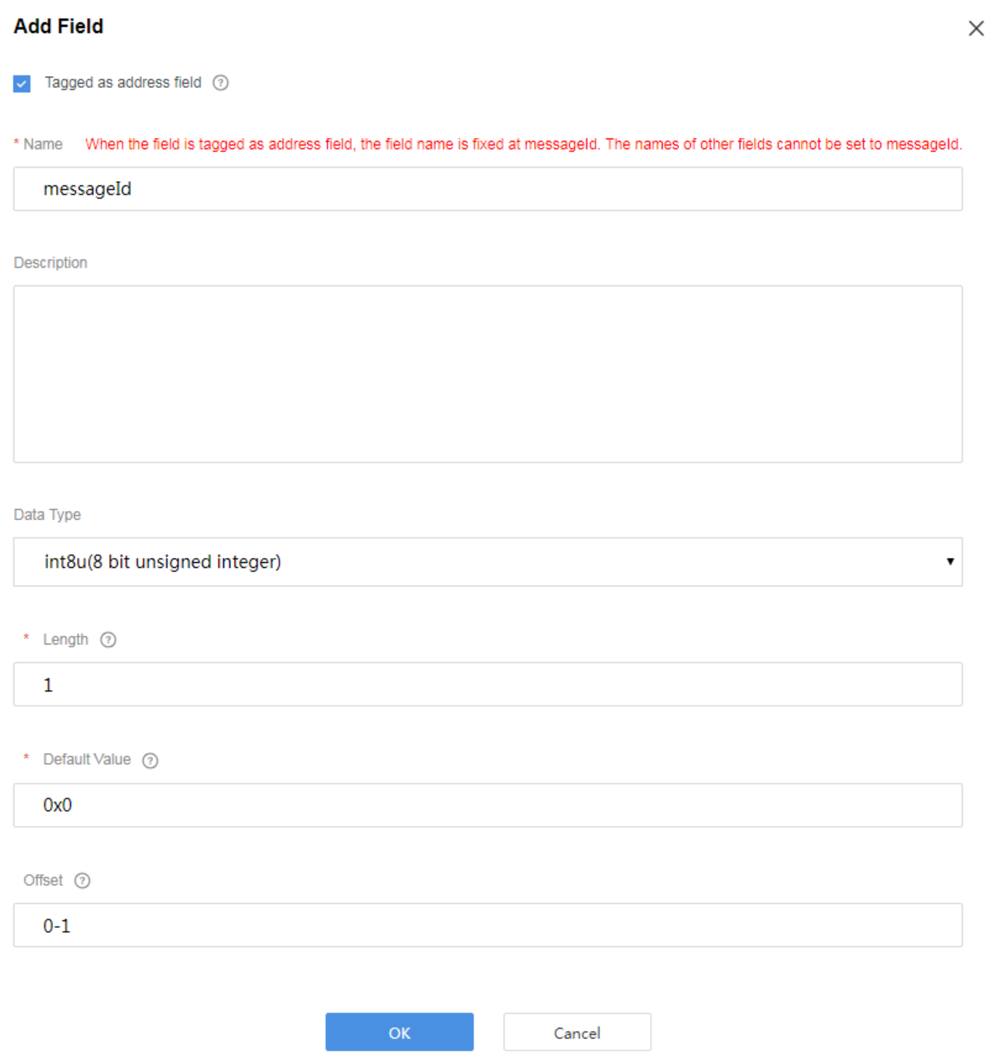
Filed

+ Add Filed

OK Cancel

Add the **messageId** field to indicate the message type.

- In this scenario, there are two types of data reporting messages. Therefore, the **messageId** field must be defined to identify the message type.
- **Data Type** is configured based on the number of data reporting message types. In this scenario, only two types of data reporting messages are available. Therefore, the value **int8u** will suffice.
- **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. In this scenario, the value **0x0** is used to identify the message that reports the fire severity and temperature.


Add Field

Tagged as address field

* Name When the field is tagged as address field, the field name is fixed at messageId. The names of other fields cannot be set to messageId.

messageId

Description

Data Type

int8u(8 bit unsigned integer)

* Length

1

* Default Value

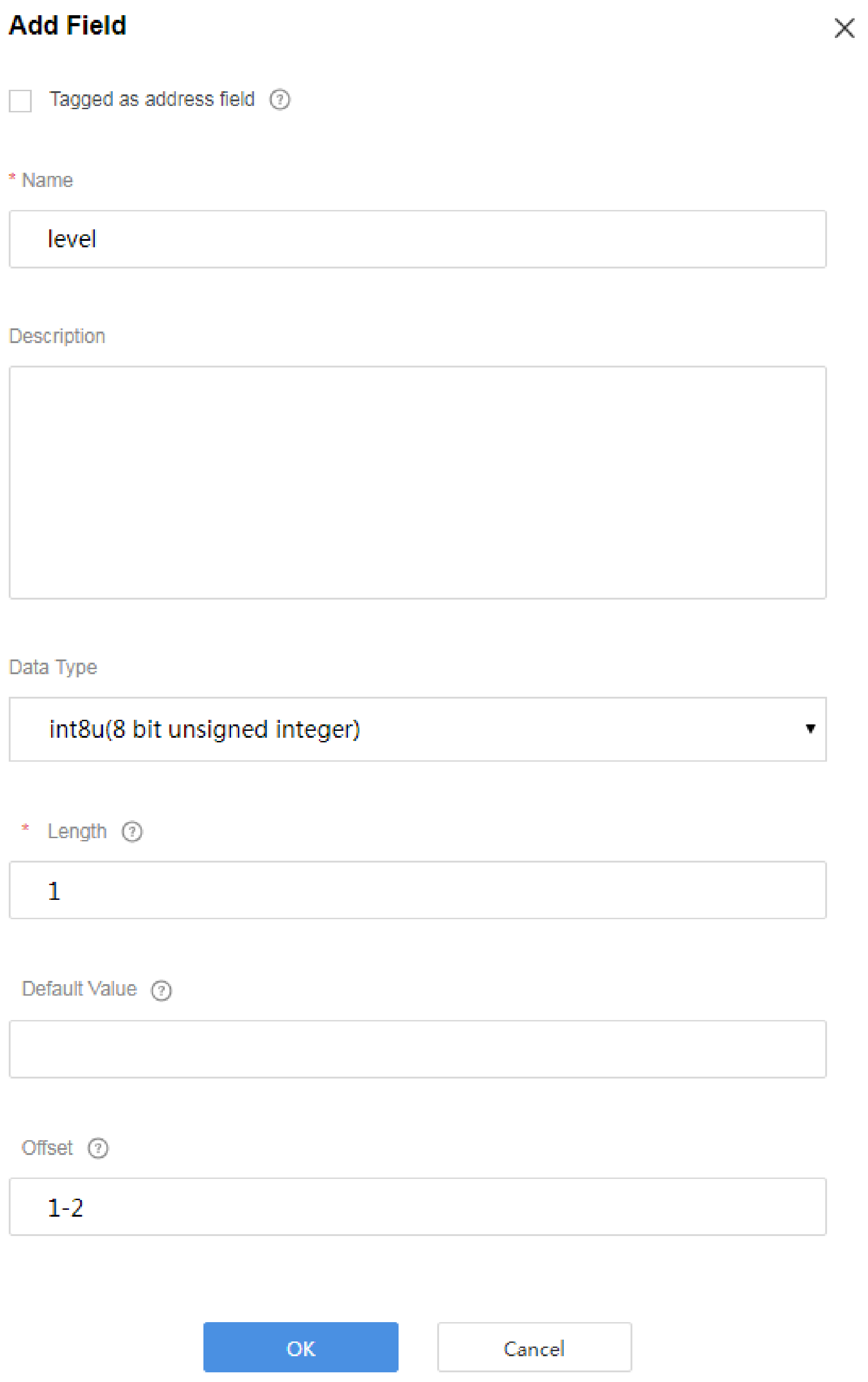
0x0

Offset

0-1

Add a **level** field to indicate the fire severity.

- **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 profile file.
- The values of **Length** and **Offset** are automatically filled based on **Data Type**.


Add Field

☐ Tagged as address field

* Name: level

Description:

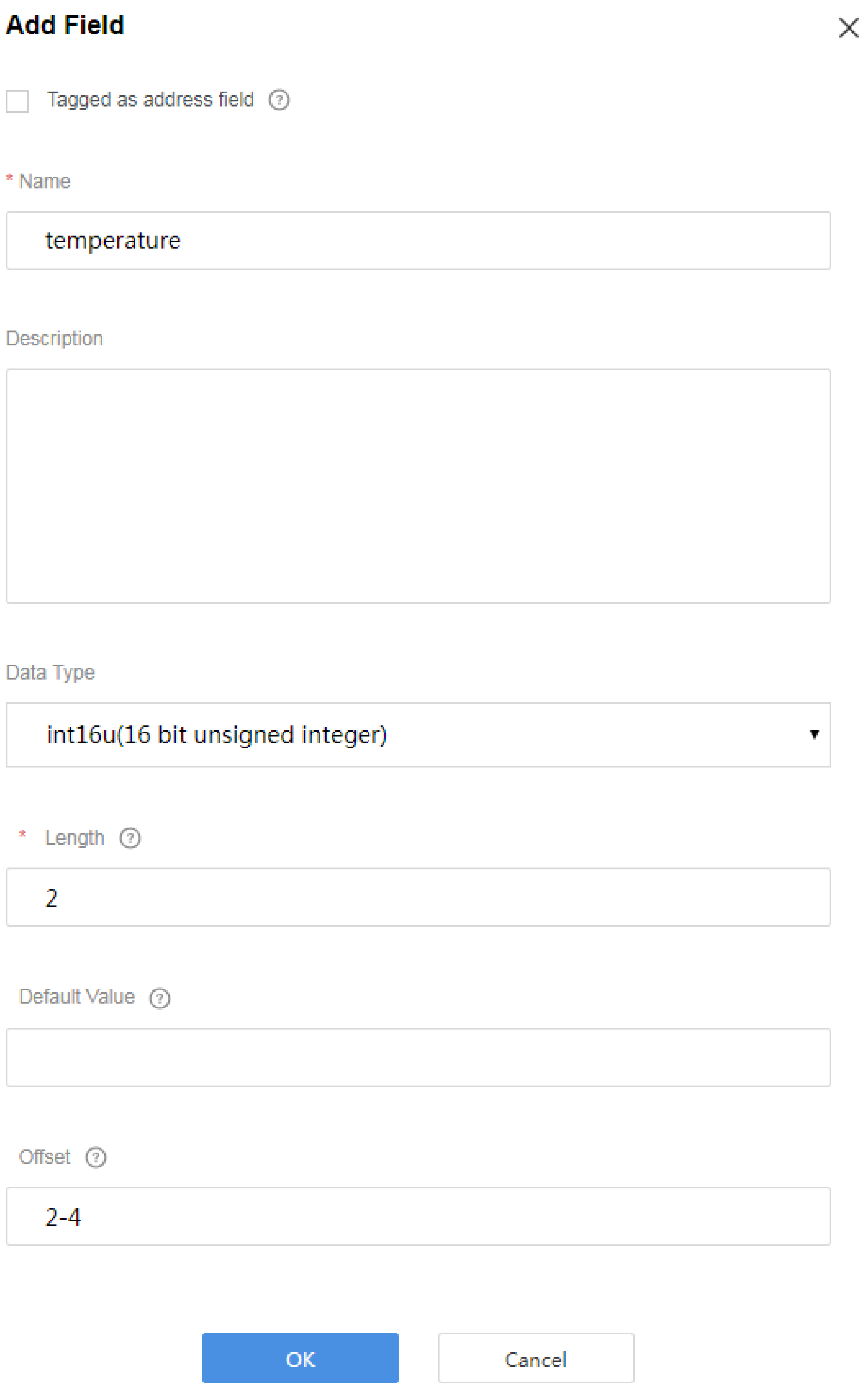
Data Type: int8u(8 bit unsigned integer)

* Length: 1

Default Value:

Offset: 1-2

Add the **temperature** field to indicate the temperature at the fire scene. In the profile file, the maximum value of **temperature** is **1000**. Therefore, set the data type of the **temperature** field to **int16u** in the codec to meet the value range requirement of **temperature**.


Add Field

Tagged as address field

* Name

temperature

Description

Data Type

int16u(16 bit unsigned integer)

* Length

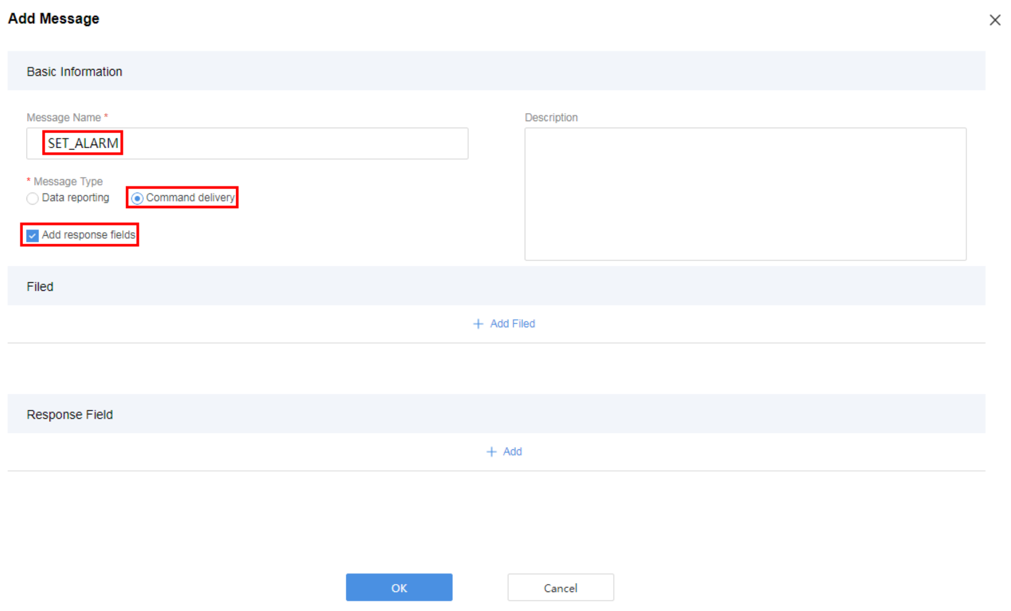
2

Default Value

Offset

2-4

Step 3 Configure a command delivery message.


Add Message

Basic Information

Message Name * SET_ALARM

Description

* Message Type
Data reporting [✓ Command delivery]

[✓ Add response fields]

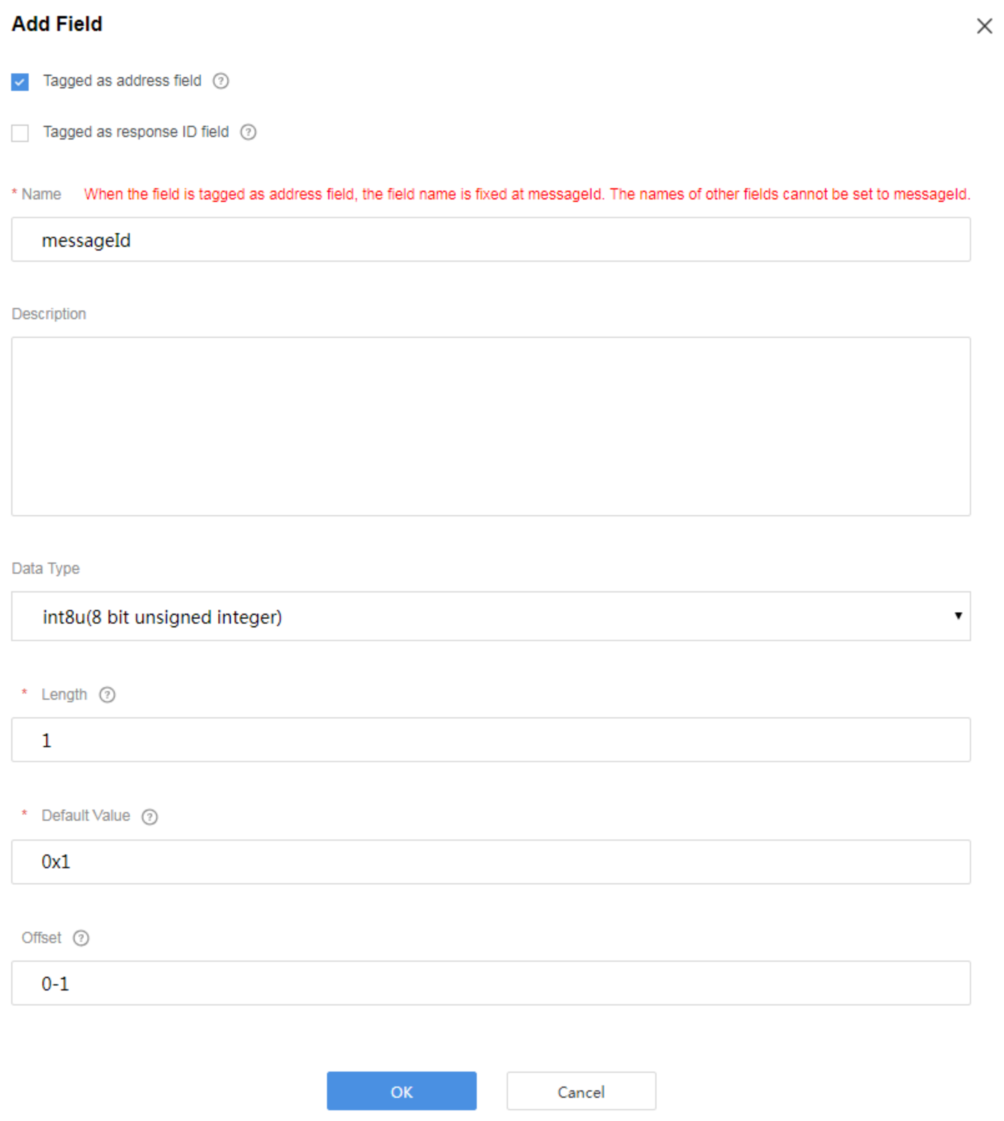
Filed

Add Filed

Response Field

Add

Add the **messageId** field to indicate the message type. If there is only one type of command delivery message, this parameter does not need to be set.



Add Field

[✓] Tagged as address field

[ ] Tagged as response ID field

* Name When the field is tagged as address field, the field name is fixed at messageId. The names of other fields cannot be set to messageId.

messageId

Description

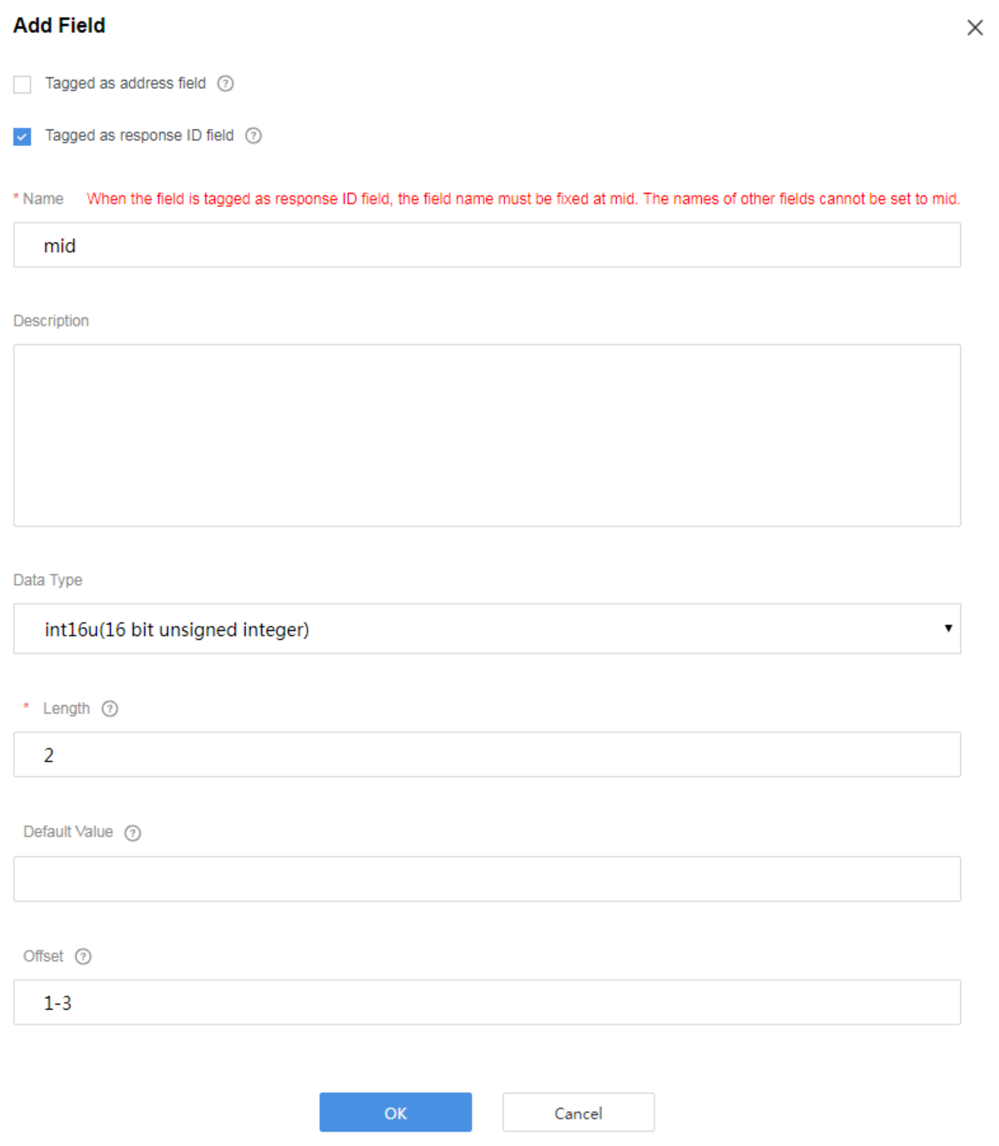
Data Type: int8u(8 bit unsigned integer)

* Length: 1

* Default Value: 0x1

Offset: 0-1

Add the **mid** field to associate the delivered command with the command execution result.


Add Field

☐ Tagged as address field

☑ Tagged as response ID field

* Name When the field is tagged as response ID field, the field name must be fixed at mid. The names of other fields cannot be set to mid.

mid

Description

Data Type

int16u(16 bit unsigned integer)

* Length

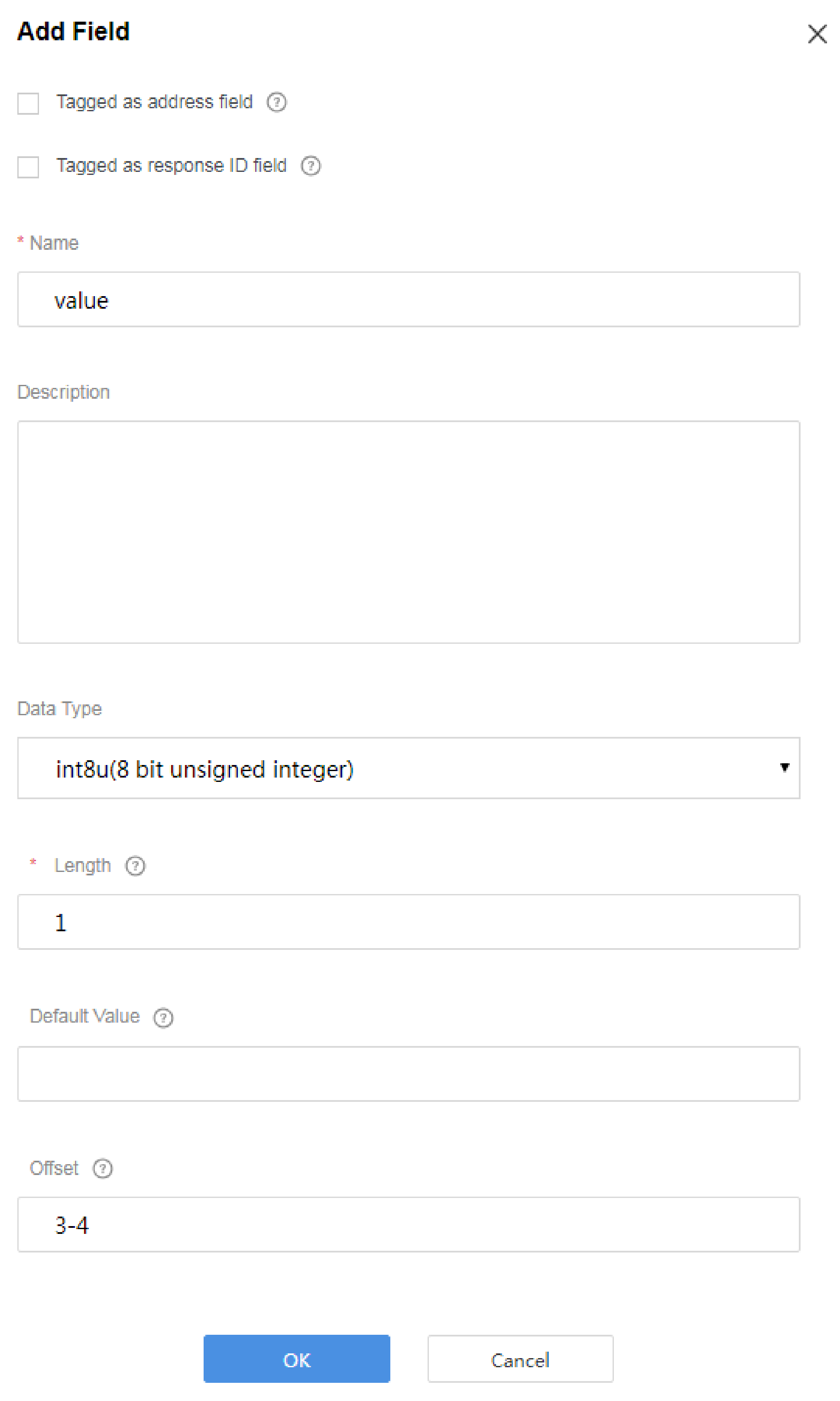
2

Default Value

Offset

1-3

Add the **value** field to indicate the parameter value of the delivered command.


Add Field

☐ Tagged as address field

☐ Tagged as response ID field

* Name: value

Description:

Data Type: int8u(8 bit unsigned integer)

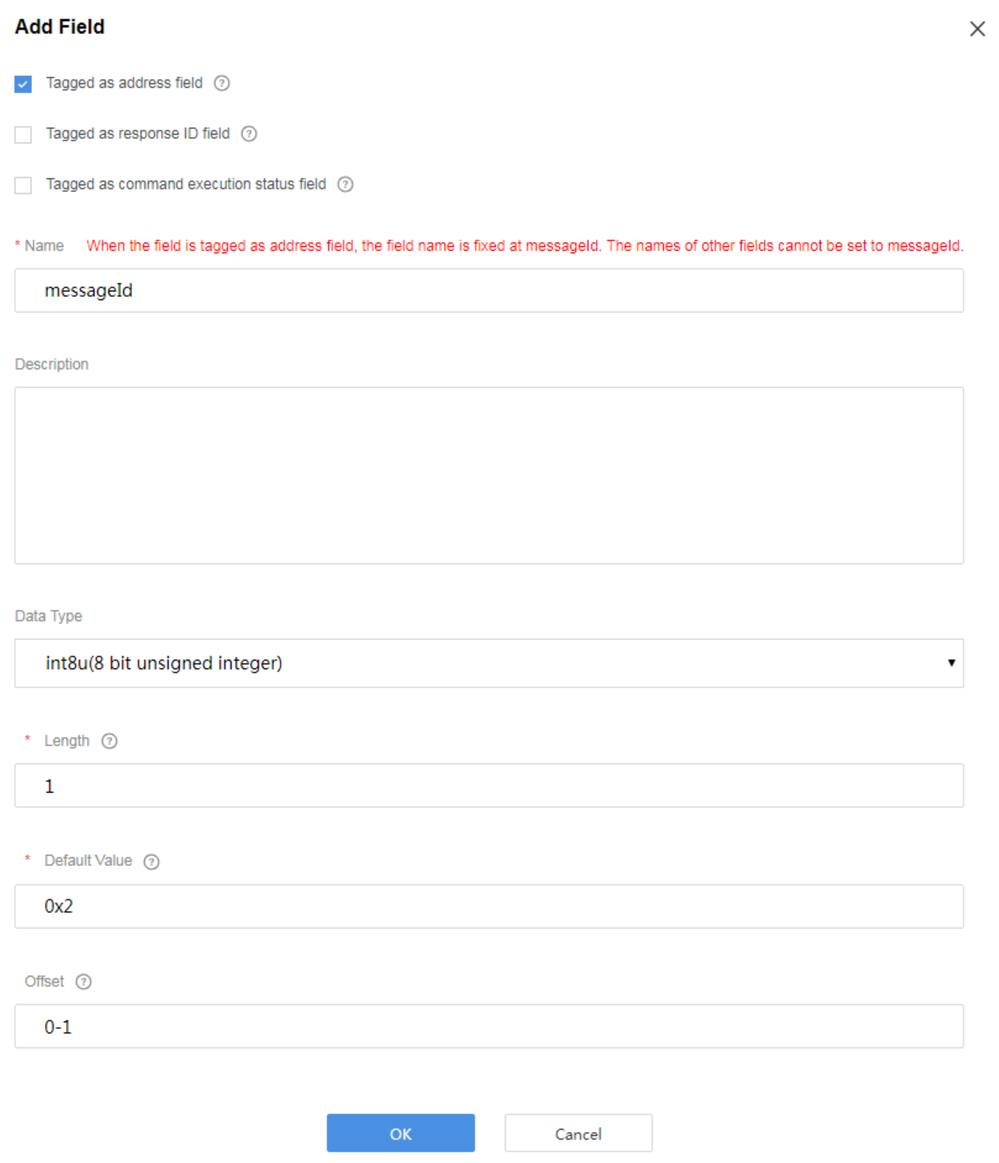
* Length: 1

Default Value:

Offset: 3-4

Step 4 Configure a command response.

Add the **messageId** field to indicate the message type. The command execution result is an upstream message, which is differentiated from the data reporting message by the **messageId** field.



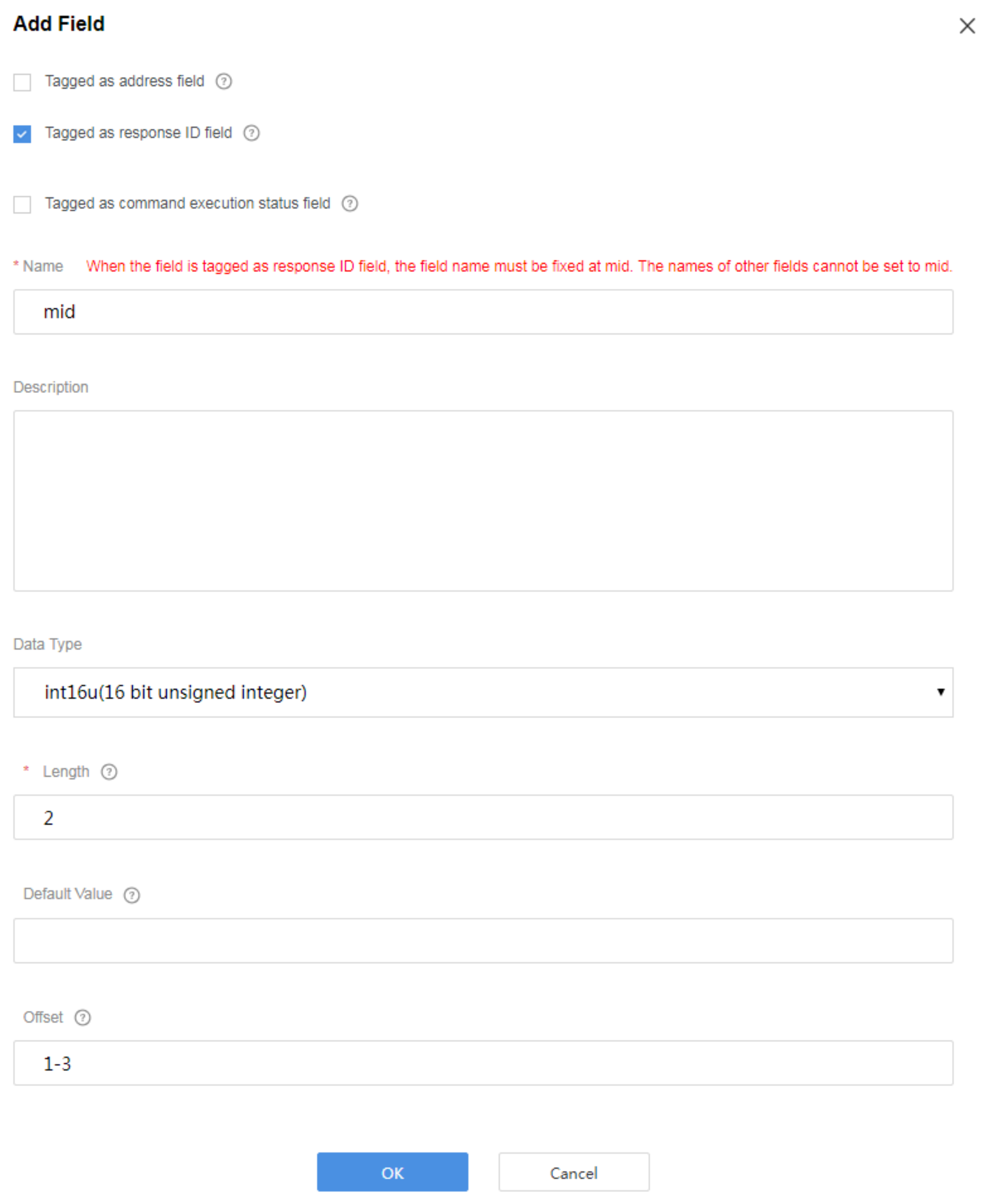
Add Field

- [✓] Tagged as address field
- [ ] Tagged as response ID field
- [ ] Tagged as command execution status field

* Name: When the field is tagged as address field, the field name is fixed at messageId. The names of other fields cannot be set to messageId.

* Name	messageId
Description	
Data Type	int8u(8 bit unsigned integer)
* Length	1
* Default Value	0x2
Offset	0-1

Add the **mid** field to associate the delivered command with the command execution result.


Add Field

- [ ] Tagged as address field
- [x] Tagged as response ID field
- [ ] Tagged as command execution status field

* Name When the field is tagged as response ID field, the field name must be fixed at mid. The names of other fields cannot be set to mid.

mid

Description

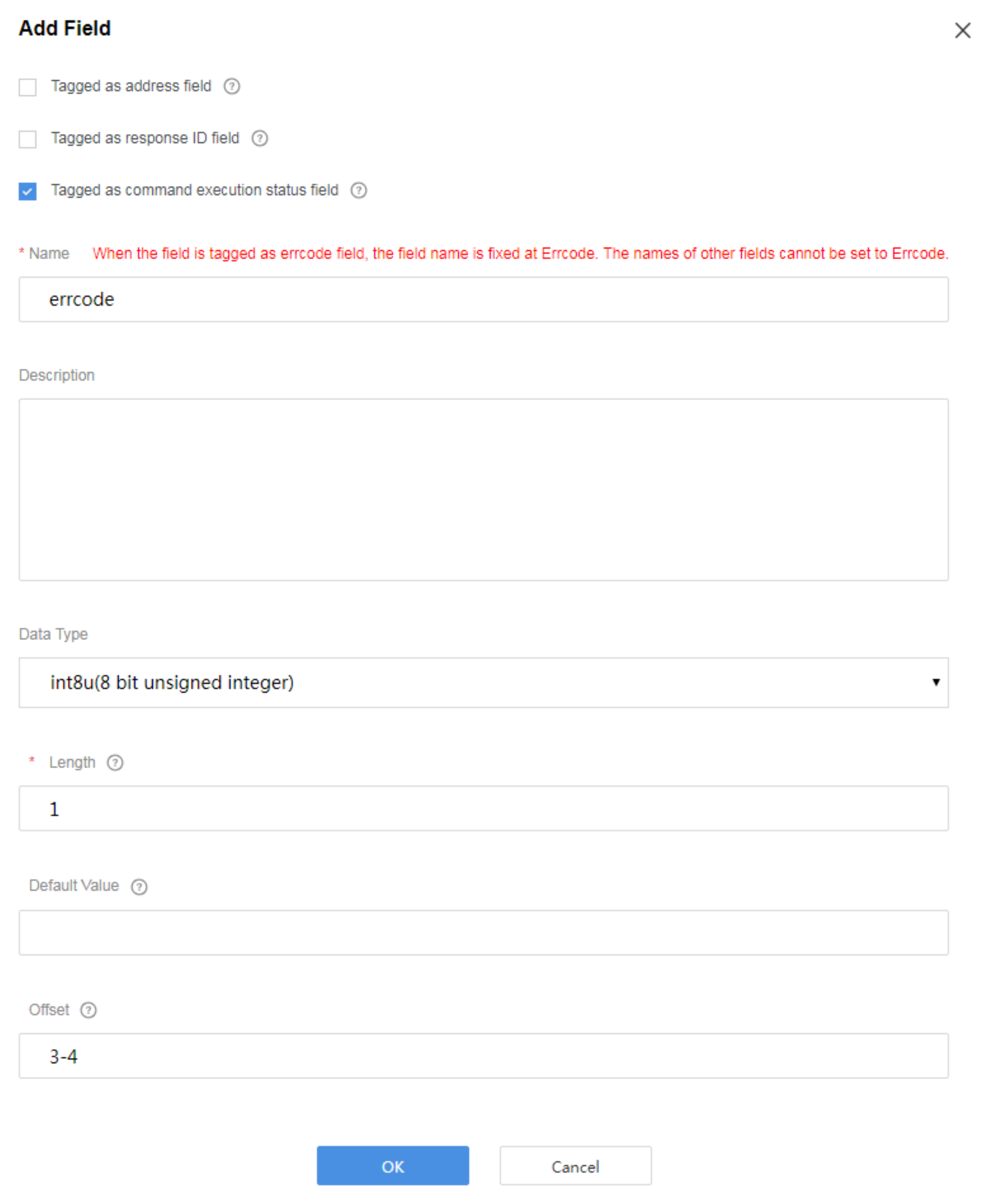
Data Type: int16u(16 bit unsigned integer)

* Length: 2

Default Value:

Offset: 1-3

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.



Add Field

☐ Tagged as address field

☐ Tagged as response ID field

☑ Tagged as command execution status field

* Name — When the field is tagged as errcode field, the field name is fixed at Errcode. The names of other fields cannot be set to Errcode.

errcode

Description

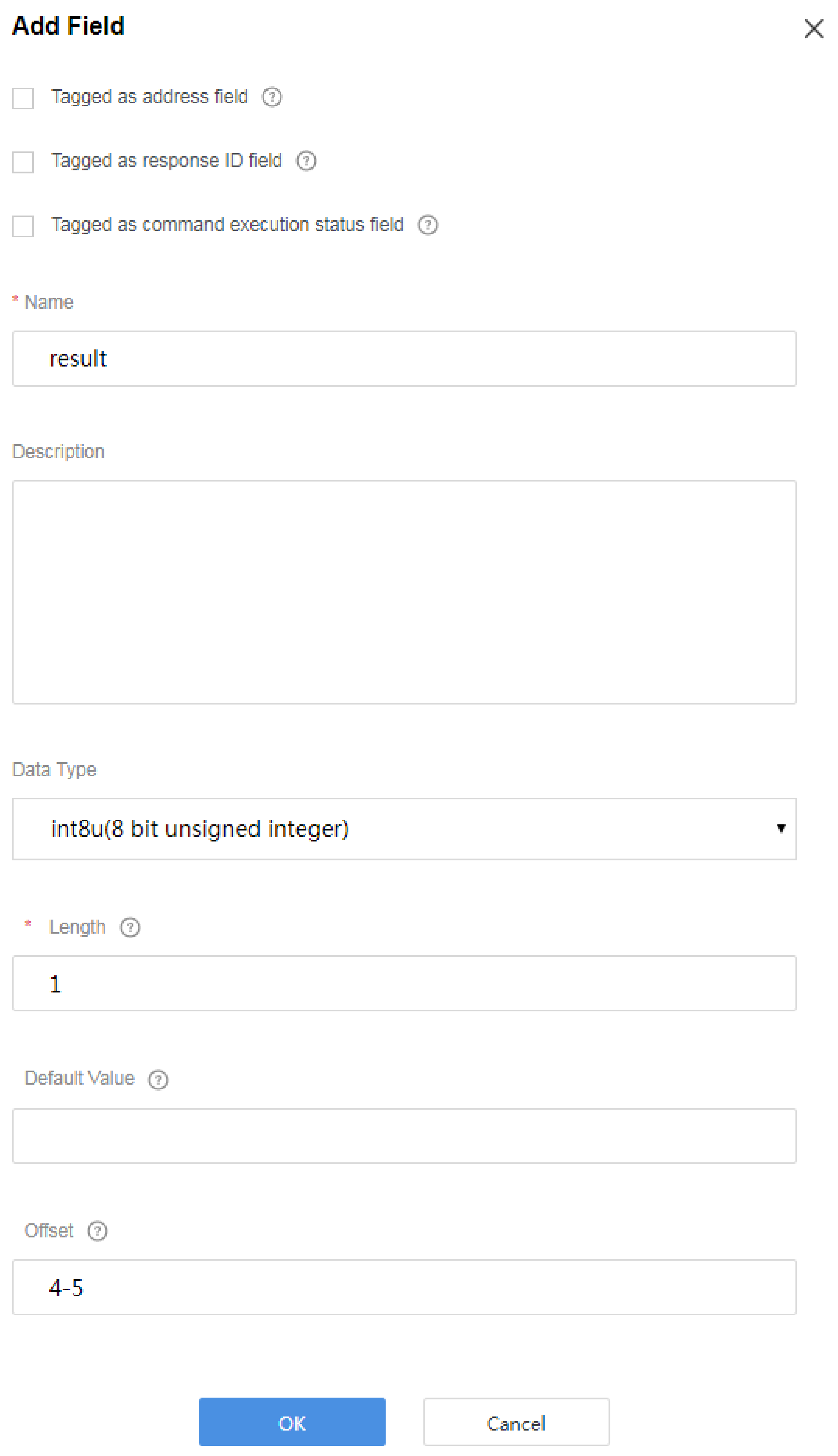
Data Type: int8u(8 bit unsigned integer)

* Length: 1

Default Value:

Offset: 3-4

Add the **result** field to indicate the command execution result.


Add Field

Tagged as address field
Tagged as response ID field
Tagged as command execution status field

* Name: result

Description:

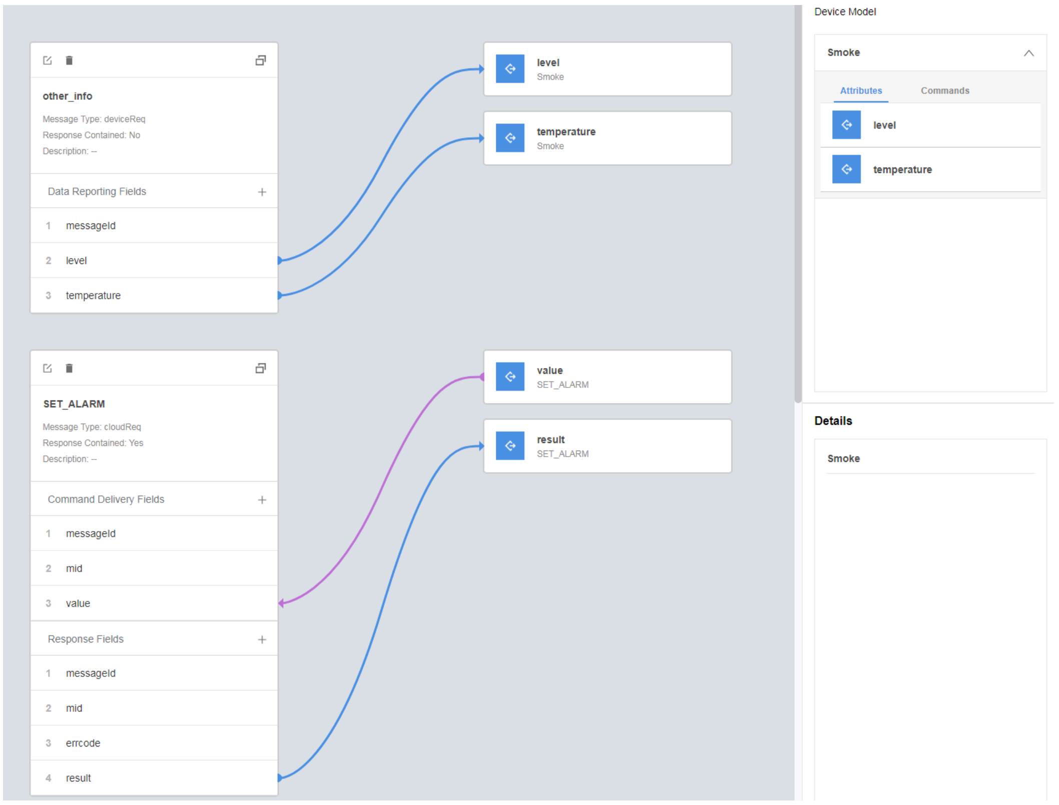
Data Type: int8u(8 bit unsigned integer)

* Length: 1

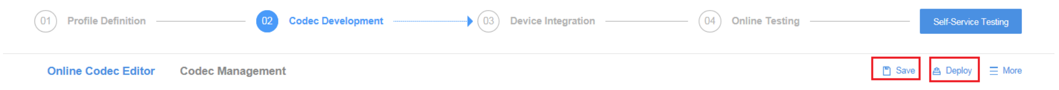
Default Value:

Offset: 4-5

Step 5 Drag the property fields and command fields in **Device Model** on the right to set up a mapping with the fields in the data reporting message and command delivery message.



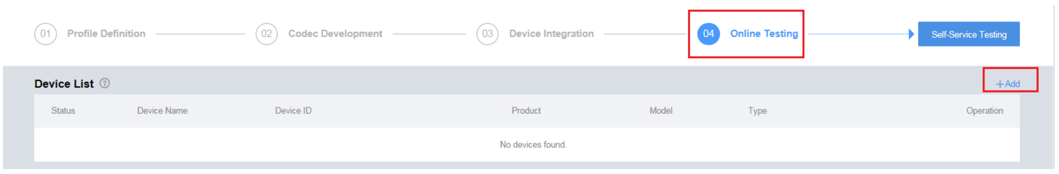
Step 6 Click **Save** and then **Deploy** to deploy the codec on the IoT platform.



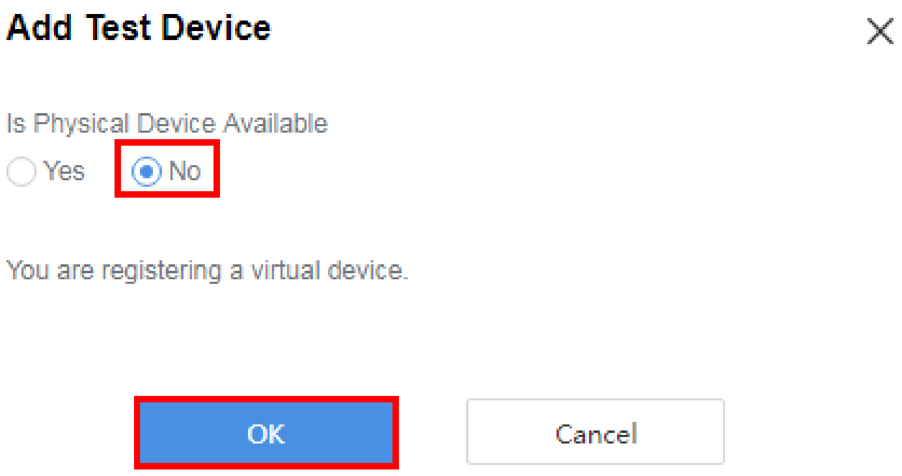
----End

Testing the Codec

Step 1 In the development space of the smoke sensor, click **Online Testing** and add a virtual device to test the codec.



Select **No** for **Is Physical Device Available** and click **OK**.


Add Test Device

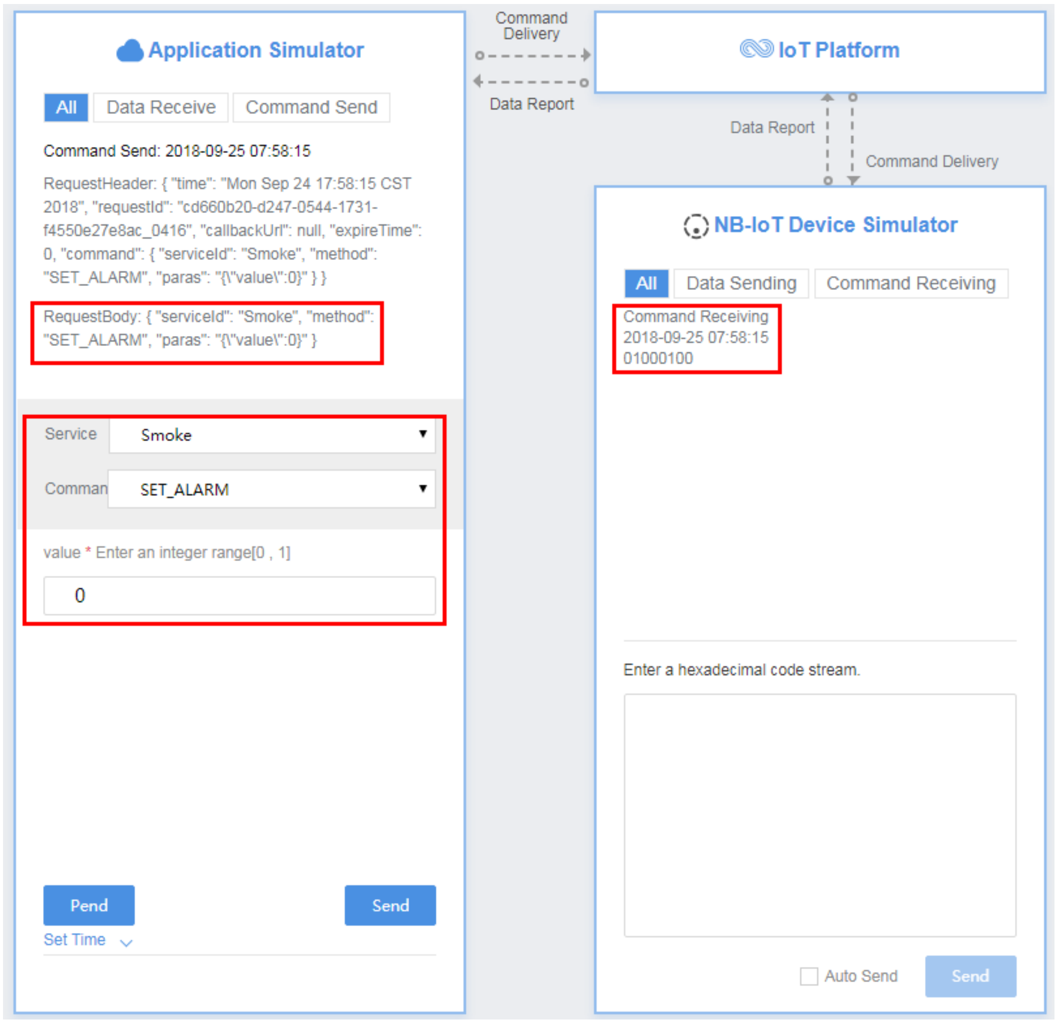
Is Physical Device Available

Yes [✓ No]

You are registering a virtual device.

Step 2 Use the application simulator to deliver a command ({ "serviceId": "Smoke", "method": "SET_ALARM", "paras": "{\"value\":0}" }).

View the command receiving result in **Device Simulator**, which is **01000100**. **01** indicates the **messageId** field, **0001** indicates the **mid** field, and **00** indicates the **value** field.



Command Send: 2018-09-25 07:58:15

RequestHeader: { "time": "Mon Sep 24 17:58:15 CST 2018", "requestId": "cd660b20-d247-0544-1731-f4550e27e8ac_0416", "callbackUrl": null, "expireTime": 0, "command": { "serviceId": "Smoke", "method": "SET_ALARM", "paras": "{\"value\":0}" } }

RequestBody: { "serviceId": "Smoke", "method": "SET_ALARM", "paras": "{\"value\":0}" }

Service: Smoke

Comman: SET_ALARM

value * Enter an integer range[0 , 1]

0

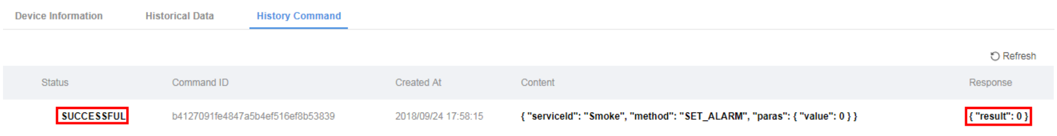
Command Receiving
2018-09-25 07:58:15
01000100

Enter a hexadecimal code stream.

Step 3 Use the device simulator to report data.

For example, a hexadecimal code stream (0200010000) is reported. In this code stream, **02** indicates the **messageId** field and specifies that this message reports the command execution result. **0001** indicates the **mid** field and its length is two bytes. **00** indicates the command execution status and its length is one byte. The second **00** indicates the command execution result and its length is one byte.

Choose **Device Management** and select the device that reports the command execution result. On the page displayed, click the **Historical Commands** tab to view the command execution status. In this case, the status is **SUCCESSFUL**.



Status	Command ID	Created At	Content	Response
SUCCESSFUL	b4127091fe4847a5b4ef516ef8b53839	2018/09/24 17:58:15	{ "serviceId": "Smoke", "method": "SET_ALARM", "paras": { "value": 0 } }	{ "result": 0 }

----End

Summary

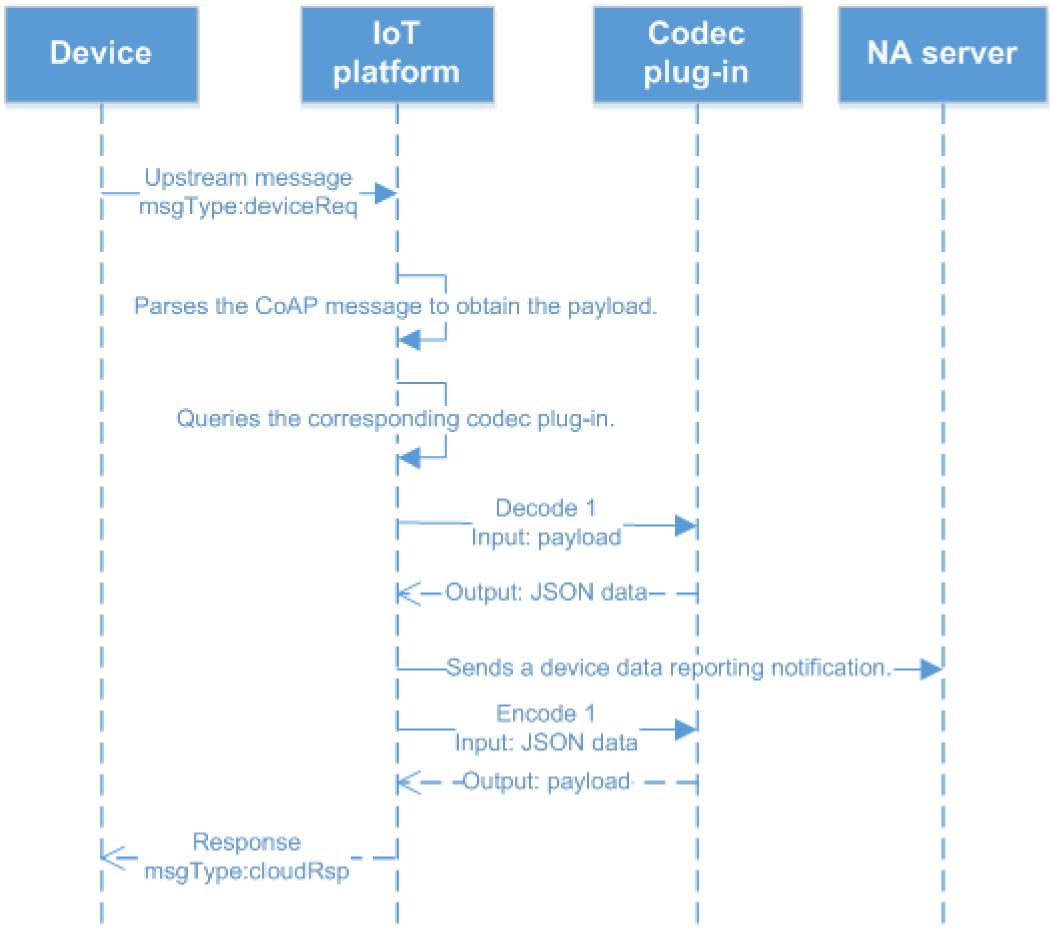
- If the codec needs to parse the command execution result, the **mid** field must be defined in the command and the command response.
- The length of the **mid** field in a command is two bytes. For each device, **mid** increases from 1 to 65535, and the corresponding code stream ranges from 0001 to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

1.4.4 Reference

1.4.4.1 Message Processing Flow

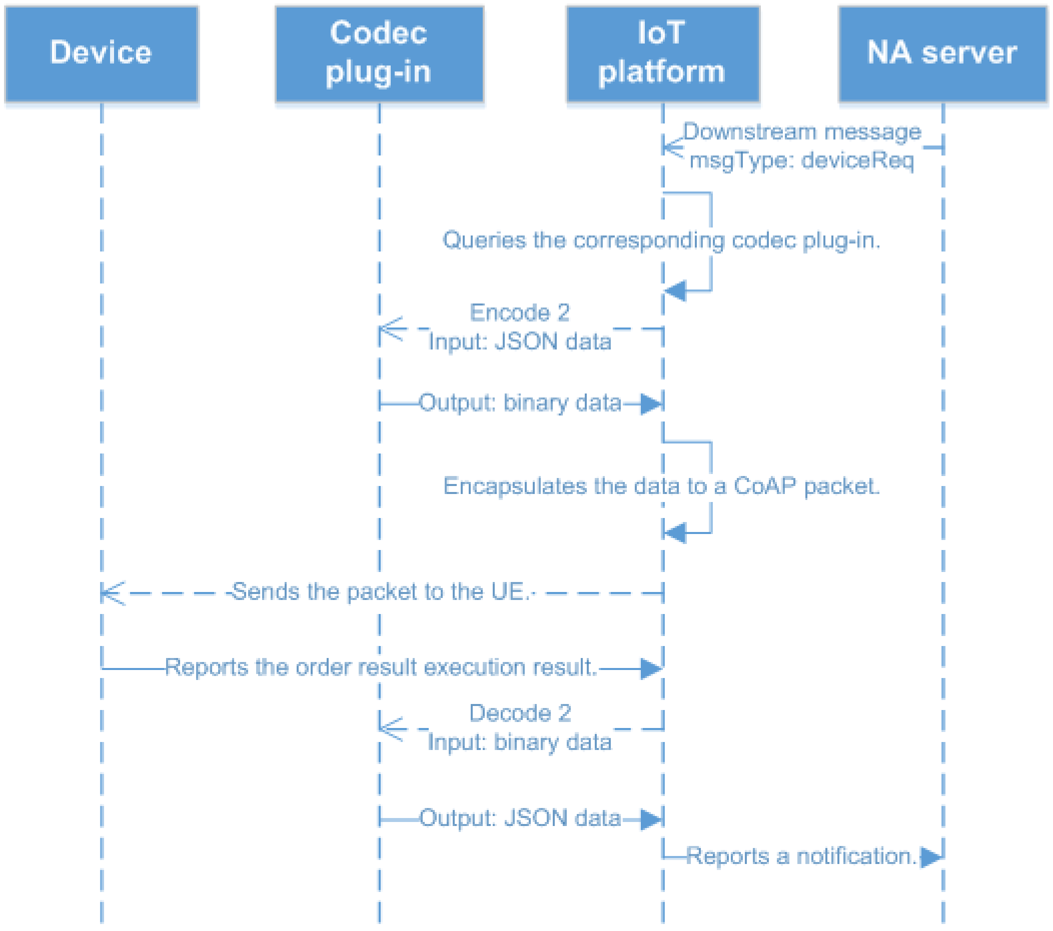
Data Reporting

Figure 1-19 Data reporting flow


Device
IoT platform
Codec plug-in
NA server
Upstream message msgType:deviceReq
Parses the CoAP message to obtain the payload.
Queries the corresponding codec plug-in.
Decode 1 Input: payload
Output: JSON data
Sends a device data reporting notification.
Encode 1 Input: JSON data
Output: payload
Response msgType:cloudRsp

Order Delivery

Figure 1-20 Order delivery flow


Device
Codec plug-in
IoT platform
NA server
Downstream message msgType: deviceReq
Queries the corresponding codec plug-in.
Encode 2 Input: JSON data
Output: binary data
Encapsulates the data to a CoAP packet.
Sends the packet to the UE.
Reports the order result execution result.
Decode 2 Input: binary data
Output: JSON data
Reports a notification.

1.4.4.2 decode API Description

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

Upstream packets of a device can be classified into the following types: data reported by device and responses of the device to the IoT platform (corresponding to messages **1** and **5** in the following figure). Message **4** is the protocol ACK message returned by the module. No plug-in processing is required. The decoding output fields vary depending on the upstream packet.

Figure 1-21 Upstream packet

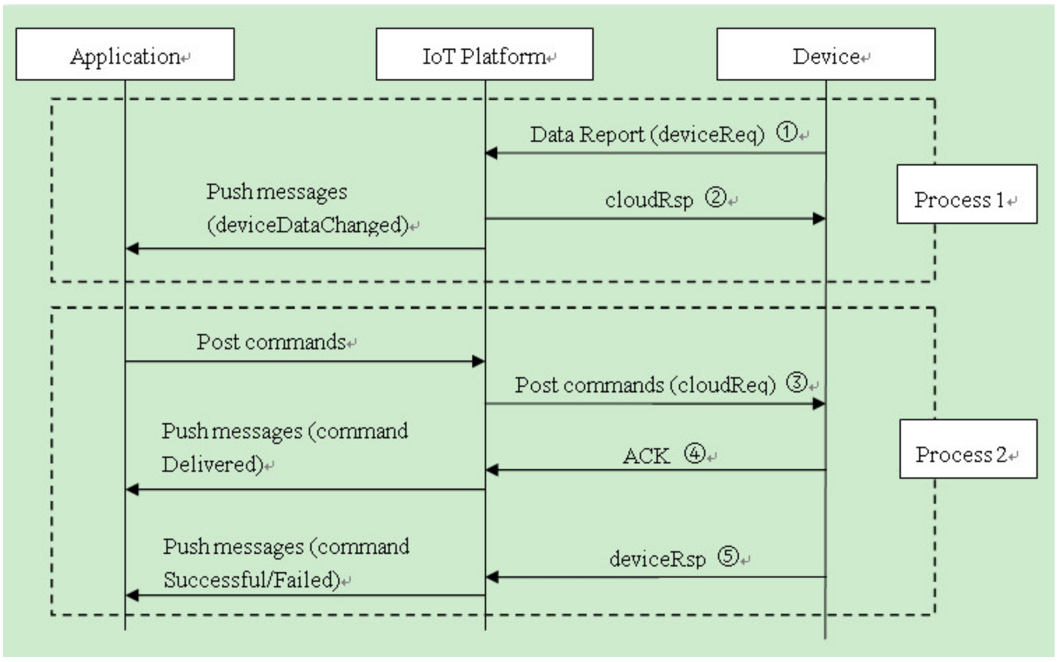

Application
IoT Platform
Device
Data Report (deviceReq) ①
Push messages (deviceDataChanged)
cloudRsp ②
Process 1
Post commands
Post commands (cloudReq) ③
Push messages (command Delivered)
ACK ④
Process 2
Push messages (command Successful/Failed)
deviceRsp ⑤

Table 1-4 Data reported by the device

Field	Type	Description	Mandatory or Optional
identifier	String	Specifies the identifier of the device in the application protocol. The IoT platform obtains the parameter over the decode interface, encodes the parameter over the encode interface, and places the parameter in a stream.	Optional
msgType	String	This field has a fixed value of **deviceReq**, which indicates that the device reports data to the IoT platform.	Mandatory

Field	Type	Description	Mandatory or Optional
hasMore	Int	Specifies whether the device has subsequent data to report. ● **0**: The device has subsequent data to report. ● **1**: The device has no subsequent data to report. Subsequent data indicates that a piece of data reported by a device may be reported in multiple times. After the data is reported in 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 assuming that the **hasMore** field is set to **0**.	Optional
data	ArrayNode	Specifies content of data reported by the device. For details, see **Table 1-5**.	Mandatory

Table 1-5 Definition of ArrayNode

Field	Type	Description	Mandatory or Optional
serviceId	String	Identifies a service.	Mandatory
serviceData	ObjectNode	Specifies the data of a service. Detailed fields are defined in the profile file.	Mandatory
eventTime	String	Specifies the data collection time, which is in the format of yyyyMMddTHHmmssZ, for example, 20161219T114920Z.	Optional

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"}
         ]
}
}
```

Table 1-6 Response sent by the device to the IoT platform

Field	Type	Description	Mandatory or Optional
identifier	String	Specifies the 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.	Optional
msgType	String	This field has a fixed value of **deviceRsp**, which indicates that the IoT platform sends a response to the device.	Mandatory
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. Otherwise, the IoT platform cannot associate the delivered command with the command execution result (deviceRsp). As a result, the IoT platform cannot update the command delivery status (success or failure) based on the command execution result (deviceRsp).	Mandatory
errcode	Int	Specifies the request processing result code. The IoT platform determines the command delivery status based on this field. ● **0**: success ● **1**: failure	Mandatory

Field	Type	Description	Mandatory or Optional
body	ObjectNode	Specifies the response to the command sent by the IoT platform. Detailed fields are defined in the profile file. **NOTE** The body is not an array.	Optional

Example:

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

1.4.4.3 Description of encode API

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

Downstream packets of the IoT platform are classified into commands sent by the IoT platform and responses sent by the IoT platform for data reported by devices (corresponding to messages **2** and **3** in the following figure). The encoding output fields vary depending on the downstream packet.

Figure 1-22 Downstream packet

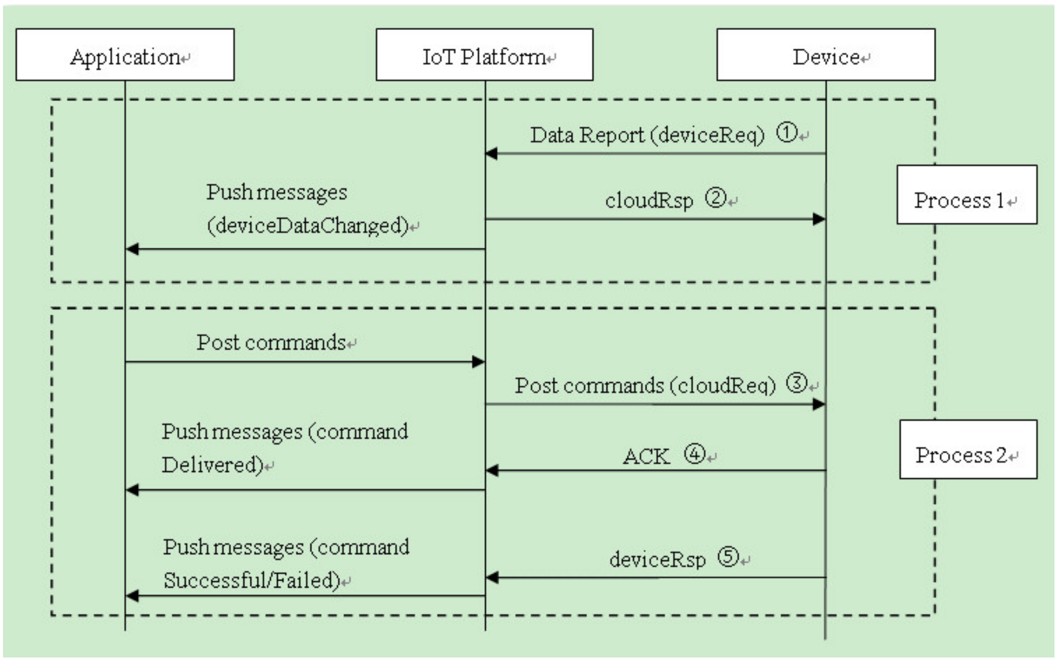

Application
IoT Platform
Device
Data Report (deviceReq) ①
Push messages (deviceDataChanged)
cloudRsp ②
Process 1
Post commands
Post commands (cloudReq) ③
Push messages (command Delivered)
ACK ④
Process 2
Push messages (command Successful/Failed)
deviceRsp ⑤

Table 1-7 Definition of input parameters of the encode API over which the IoT platform delivers commands

Field	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.	Optional
msgType	String	This field has a fixed value of **cloudReq**, which indicates that the IoT platform delivers a request.	Mandatory
serviceId	String	Identifier of a service.	Mandatory
cmd	String	Name of a service command. For details about the service command definition, see the profile file.	Mandatory
paras	ObjectNode	Command parameters. Detailed fields are defined in the profile file.	Mandatory
hasMore	Int	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.	Mandatory

Field	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.	Mandatory

Example:

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

Table 1-8 Definition of input parameters of the encode API over which the IoT platform responds to data reported by a device

Field	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.	Optional
msgType	String	This field has a fixed value of **cloudRsp**, which indicates that the IoT platform sends a response for data reported by a device.	Mandatory
request	byte[]	Data reported by the device.	Mandatory

Field	Type	Description	Mandatory or Optional
errcode	int	Request processing result code. The IoT platform determines the command delivery status based on this field. ● **0**: success ● **1**: failure	Mandatory
hasMore	int	Whether the IoT platform has subsequent messages to deliver. ● **0**: The IoT platform does not have subsequent messages to deliver. ● **1**: The IoT platform has subsequent messages to deliver. 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.	Mandatory

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
}
```

1.4.4.4 getManufacturerId Interface Description

This interface is used to return the vendor ID in the format of a character string. The IoT platform calls this interface to obtain the vendor ID to associate the codec plug-in with the profile file. The association is successful only when the vendor ID and device model are consistent.

Example:

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

1.4.4.5 getModel Interface Description

This interface is used to return the device model in the format of a character string. The IoT platform calls this interface to obtain the device model to associate the codec plug-in with the profile file. The association is successful only when the vendor ID and device model are consistent.

Example:

```
@Override
public String getModel() {
return "TestUtf8Model";
}
```

1.4.4.6 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";
return node;
}

@Override
public byte[] encode(finalObjectNode input) throws Exception {
if ("Failed".equals(status)) {
status = null;
return null;
}
byte[] output;
           ...;
status = null;
return output;
}
}
```

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

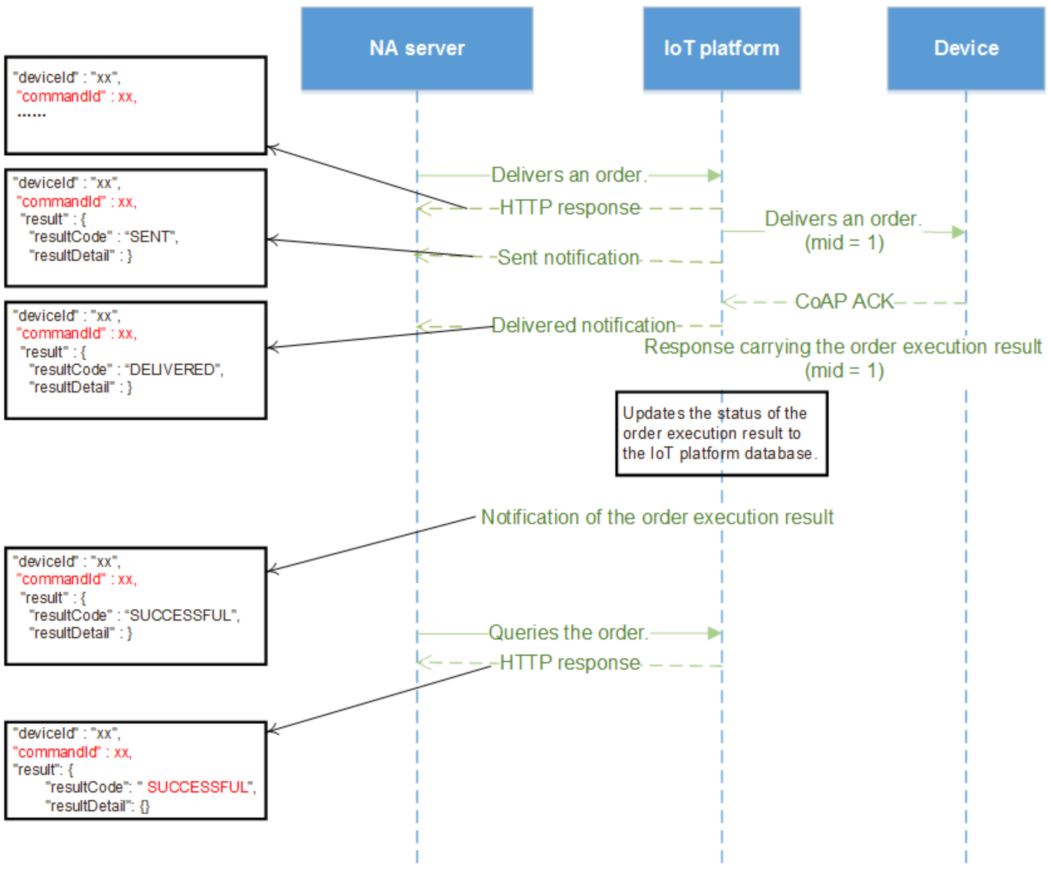
```
public class ProtocolAdapter {
@Override
public ObjectNode decode(finalbyte[] binaryData) throws Exception {
ObjectNode node;
  ...;
return node;
}

@Override
public byte[] encode(finalObjectNode input) throws Exception {
byte[] output;
  ...;
return output;
}
}
```

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:

Figure 1-23 Message flow with the MID implemented

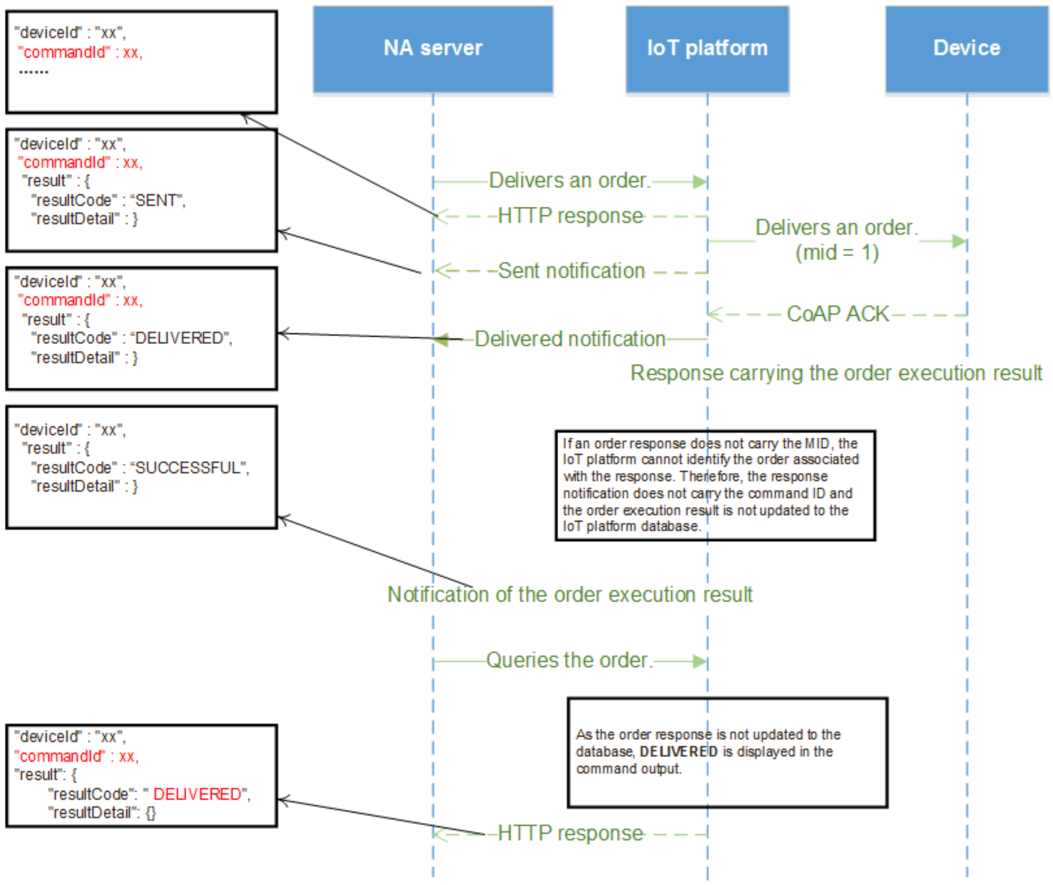


If the MID is implemented and the order execution result is reported successfully:

1. The status (**SUCCESSFUL/FAILED**) in the order execution result is updated to the record of the order in the IoT platform database.
2. The order execution result notification sent by the IoT platform to the NA server contains **commandId**.
3. The query result of the NA server indicates that the status of the order is **SUCCESSFUL/FAILED**.

When the MID is not implemented:

Figure 1-24 Message flow with the MID unimplemented



```
"deviceId" : "xx",
"commandId" : xx,
......
```

```
"deviceId" : "xx",
"commandId" : xx,
 "result" : {
   "resultCode" : "SENT",
   "resultDetail" : }
```

```
"deviceId" : "xx",
"commandId" : xx,
 "result" : {
   "resultCode" : "DELIVERED",
   "resultDetail" : }
```

```
"deviceId" : "xx",
 "result" : {
   "resultCode" : "SUCCESSFUL",
   "resultDetail" : }
```

If an order response does not carry the MID, the IoT platform cannot identify the order associated with the response. Therefore, the response notification does not carry the command ID and the order execution result is not updated to the IoT platform database.

As the order response is not updated to the database, DELIVERED is displayed in the command output.

```
"deviceId" : "xx",
"commandId" : xx,
"result": {
      "resultCode": " DELIVERED",
      "resultDetail": {}
```

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

1. The status (**SUCCESSFUL/FAILED**) in the order execution result is not updated to the record of the order in the IoT platform database.
2. The order execution result notification sent by the IoT platform to the NA server does not contain **commandId**.
3. 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 concerned but the order execution is not concerned, the device and codec plug-in do not need to process the MID.
- If the MID is not implemented after the vendor evaluation, 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.
  - Identify the mapping between the order execution result response and the delivered order according to the information in the **resultDetail** field. The codec plug-in can add order-related information, such as an order code, to the **resultDetail** field of the order response to help identify the order.

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
{
      ...
}
```

1.4.4.7 Input/Output Format of the Codec Plug-In

Table 1-9 Definition of services supported by a type of water meter

Service Type	Attribute Name	Attribute Description	Attribute Type (Data Type)
Battery	-	-	-
-	batteryLevel	Specifies the battery level in the unit of percent. The value ranges from 0 to 100.	int
Meter	-	-	-

Service Type	Attribute Name	Attribute Description	Attribute Type (Data Type)
-	signalStrength	Specifies 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
}
```

NOTE

The value of **request** can be [1,2], which is simulated data. The actual value prevails.

Table 1-10 Order definition

Basic Function	Type	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	● **0**: success ● **1**: invalid input ● **2**: 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
}
```

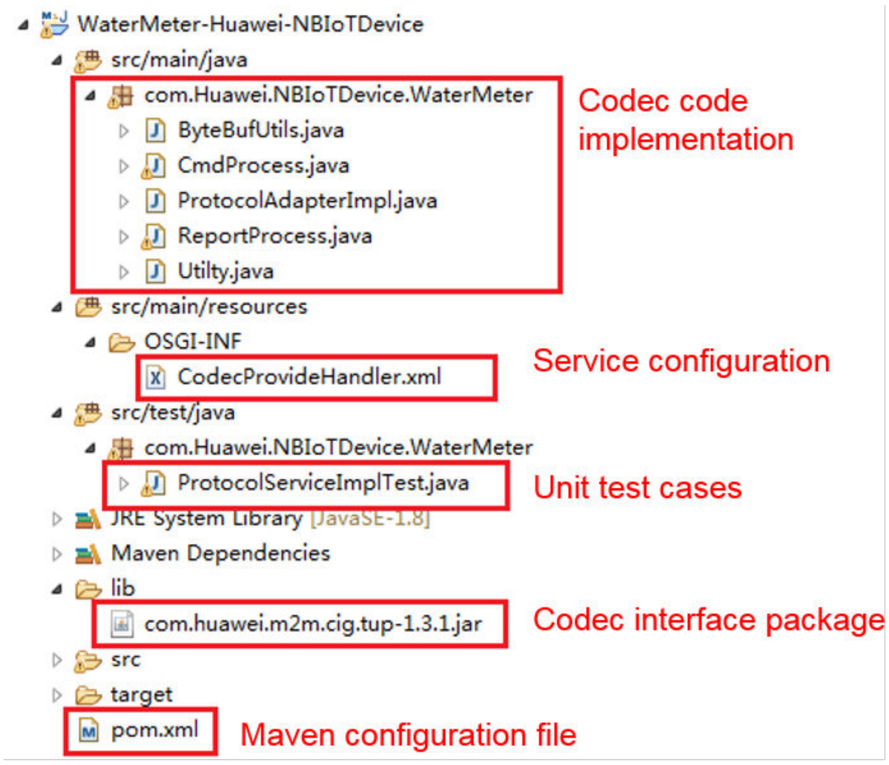
After the IoT platform receives a response from the device, the IoT platform invokes the decode interface for decoding. The decode interface output is as follows:

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

1.4.4.8 Implementation Sample Interpretation

In the DEMO project of the codec (click **here** to obtain), an example codec is provided. The following figure shows the sample project structure.

Figure 1-25 Sample project structure



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

NOTE

Use the encryption algorithms supported by the JDK. For details about these encryption algorithms, see **Appendix: Encryption Algorithms Supported by the JDK**.

- Maven configuration file

  In the **pom.xml** file, modify the name of the codec according to the naming rule.

```
<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-device model, for example: WaterMeter-Huawei-
NBIoTDevice.-->
<artifactId>WaterMeter-Huawei-NBIoTDevice</artifactId>
<version>1.0.0</version>
<!-- Check that the value is bundle. The value cannot be jar. -->
<packaging>bundle</packaging>

<properties>
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<junit.version>4.11</junit.version>
<fasterxml.jackson.version>2.7.4</fasterxml.jackson.version>
<felix.maven.plugin.version>2.5.4.fixed2</felix.maven.plugin.version>
```

```
<json.lib.version>2.4</json.lib.version>
<m2m.cig.version>1.3.1</m2m.cig.version>
<slf4j.api.version>1.7.6</slf4j.api.version>
</properties>

<dependencies>
<!-- Used by unit test -->
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>${junit.version}</version>
</dependency>
<!-- Used by logs -->
<dependency>
<groupId>org.slf4j</groupId>
<artifactId>slf4j-api</artifactId>
<version>${slf4j.api.version}</version>
</dependency>
<!-- Used for converting JSON; mandatory -->
<dependency>
<groupId>com.fasterxml.jackson.core</groupId>
<artifactId>jackson-databind</artifactId>
<version>${fasterxml.jackson.version}</version>
</dependency>
<!-- Codec API provided by Huawei; mandatory -->
<!-- Replace systemPath with your local \codecDemo\lib
\com.huawei.m2m.cig.tup-1.3.1.jar -->
<dependency>
<groupId>com.huawei</groupId>
<artifactId>protocal-jar</artifactId>
<version>1.3.1</version>
<scope>system</scope>
<systemPath>${basedir}/lib/com.huawei.m2m.cig.tup-1.3.1.jar</systemPath>
</dependency>

<!-- In this example, the JAR file used for data conversion is written here.
Enter artifactId in the Embed-Dependency. -->
<dependency>
<groupId>net.sf.json-lib</groupId>
<artifactId>json-lib</artifactId>
<version>2.4</version>
<classifier>jdk15</classifier>
</dependency>

</dependencies>
<build>
<plugins>
<!-- The JDK1.8 version must be used. -->
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<configuration>
<source>1.8</source>
<target>1.8</target>
</configuration>
</plugin>
<!-- OSGi packaging configuration -->
<plugin>
<groupId>org.apache.felix</groupId>
<artifactId>maven-bundle-plugin</artifactId>
<version>${felix.maven.plugin.version}</version>
<extensions>true</extensions>
<configuration>
<buildDirectory>./target</buildDirectory>
<archive>
<addMavenDescriptor>false</addMavenDescriptor>
</archive>
<instructions>
<Bundle-RequiredExecutionEnvironment>J2SE-1.5</Bundle-
```

```
RequiredExecutionEnvironment>
<Bundle-Activator></Bundle-Activator>
<Service-Component>OSGI-INF/*</Service-Component>
<!-- Change it to the name of your codec. The naming rule is as follows:
device type-manufacturer ID-device model, for example: WaterMeter-Huawei-
NBIoTDevice. -->
<Bundle-SymbolicName>WaterMeter-Huawei-NBIoTDevice</Bundle-SymbolicName>
<Export-Package></Export-Package>
<!-- Import packages in the code and use commas (,) to separate them. [JAR
packages that start with java.** and that are referenced in Embed-Dependency
do not need to be imported in Import-Package. Otherwise, the codec cannot be
started.] -->
<Import-Package>
org.slf4j,
org.slf4j.spi,
org.apache.log4j.spi,
com.huawei.m2m.cig.tup.modules.protocol_adapter,
com.fasterxml.jackson.databind,
com.fasterxml.jackson.databind.node
</Import-Package>
<!-- For all dependency packages except junit, slf4j-api, jackson-databind,
and protocol-jar, set artifactId of each package to Embed-Dependency.
Separate artifactId values by commas (,). During Maven packaging, pack your
dependency packages into your JAR package. -->
<Embed-Dependency>
json-lib
</Embed-Dependency>
</instructions>
</configuration>
<executions>
<execution>
<id>generate-resource</id>
<goals>
<goal>manifest</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
</build>
</project>
```

- Codec code implementation
  - In the **ProtocolAdapterImpl.java** file, change the values of **MANU_FACTURERID** and **MODEL**. The IoT platform associates the codec with the profile file using the manufacturer ID and device model.

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

  - 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;
        }
    }

}
```

– 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 encoded.
                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;
        }
    }
}
```

1.4.4.9 Appendix: Encryption Algorithms Supported by the JDK

Digest Algorithm

Algorithm Name	Algorithm	Hash Length	Remarks
MD	MD2	128	-
	MD5	128	-
SHA	SHA-1	160	-
	SHA-256	256	-
	SHA-384	384	-
	SHA-512	512	-
HMAC	HmacMD5	128	-
	HmacSHA1	160	-
	HmacSHA256	256	-
	HmacSHA384	384	-
	HmacSHA512	512	-

Symmetric Encryption Algorithm

Algorithm Name	Key Length	Default Length	Work Mode	Padding Mode	Remarks
DES	56	56	ECB, CBC, PCBC, CTR, CTS, CFB, CFB8 to 128, OFB, and OFB8 to 128	NoPadding, PKCS5Padding, and ISO10126Padding	-
3DES	112 or 168	168	ECB, CBC, PCBC, CTR, CTS, CFB, CFB8 to 128, OFB, and OFB8 to 128	NoPadding, PKCS5Padding, and ISO10126Padding	-
AES	128, 192, or 256	128	ECB, CBC, PCBC, CTR, CTS, CFB, CFB8 to 128, OFB, and OFB8 to 128	NoPadding, PKCS5Padding, and ISO10126Padding	The 256-bit key needs to obtain the permission file without policy restriction.

Asymmetric Encryption Algorithm

Algorithm Name	Key Length	Default Length	Work Mode	Padding Mode	Remarks
DH	512-1024 (a multiple of 64)	1024	N/A	N/A	-

Base64 is also supported by the JDK.

1.5 Developing an Application

1.5.1 Application Connection to the IoT Platform

Overview

An NA needs to call the authentication API to connect to the IoT platform. For details about the authentication API, see the API reference document.

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

Prerequisites

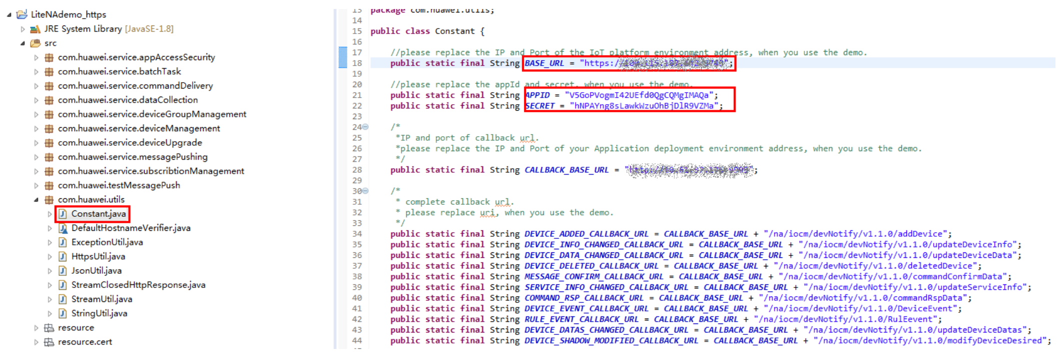
- The codec has been deployed on the IoT platform. If **Data Type** of the device is **JSON**, codec development is not required.
- If HTTPS is used for API calling, related certificates have been uploaded to the IoT platform. For details, see Resources.
- You have obtained the **Java code sample for calling the APIs**. You have also configured the development environment and imported the code sample by following the instructions provided in **Preparing the Java Development Environment**.

Procedure

Step 1 Prepare the Java development environment by following the instructions provided in **Preparing the Java Development Environment**.

This document uses the operations in Java development environment as an example.

Step 2 In the Eclipse, choose **src** > **com.huawei.utils** > **Constant.java**, and modify the values of **BASE_URL**, **APPID**, and **SECRET**.

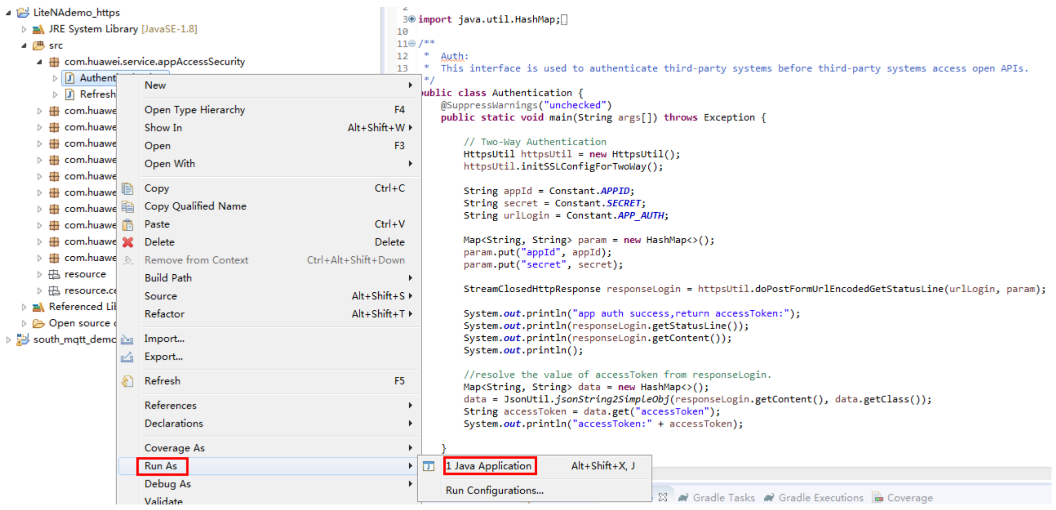

LiteNAdemo_https
JRE System Library [JavaSE-1.8]
src
com.huawei.service.appAccessSecurity
com.huawei.service.batchTask
com.huawei.service.commandDelivery
com.huawei.service.dataCollection
com.huawei.service.deviceGroupManagement
com.huawei.service.deviceManagement
com.huawei.service.deviceUpgrade
com.huawei.service.messagePushing
com.huawei.service.subscribtionManagement
com.huawei.testMessagePush
com.huawei.utils
Constant.java
DefaultHostnameVerifier.java
ExceptionUtil.java
HttpsUtil.java
JsonUtil.java
StreamClosedHttpResponse.java
StreamUtil.java
StringUtil.java
resource
resource.cert

```
package com.huawei.utils;

public class Constant {

    //please replace the IP and Port of the IoT platform environment address, when you use the demo.
    public static final String BASE_URL = "https://[redacted]";

    //please replace the appId and secret, when you use the demo.
    public static final String APPID = "V5GoPVogmI42UEfd0QgCQMgIMAQa";
    public static final String SECRET = "hNPAYng8sLawkWzuOhBjDlR9VZMa";

    /*
     *IP and port of callback url.
     *please replace the IP and Port of your Application deployment environment address, when you use the demo.
     */
    public static final String CALLBACK_BASE_URL = "[redacted]";

    /*
     * complete callback url.
     * please replace uri, when you use the demo.
     */
    public static final String DEVICE_ADDED_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/addDevice";
    public static final String DEVICE_INFO_CHANGED_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/updateDeviceInfo";
    public static final String DEVICE_DATA_CHANGED_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/updateDeviceData";
    public static final String DEVICE_DELETED_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/deletedDevice";
    public static final String MESSAGE_CONFIRM_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/commandConfirmData";
    public static final String SERVICE_INFO_CHANGED_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/updateServiceInfo";
    public static final String COMMAND_RSP_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/commandRspData";
    public static final String DEVICE_EVENT_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/DeviceEvent";
    public static final String RULE_EVENT_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/RulEevent";
    public static final String DEVICE_DATAS_CHANGED_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/updateDeviceDatas";
    public static final String DEVICE_SHADOW_MODIFIED_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/modifyDeviceDesired";
```

Parameters are described as follows:

- **BASE_URL**: Set this parameter to the application address and port number.
- **APPID**: Set this parameter to the application ID obtained after the application (or project) is created.
- **SECRET**: Set this parameter to the secret obtained after the application (or project) is created.

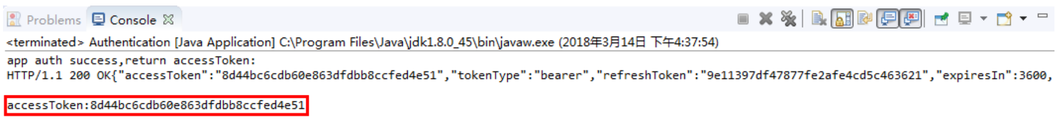
Step 3 In the Eclipse, choose **src** > **com.huawei.service.appAccessSecurity**, right-click **Authentication.java**, and choose **Run As** > **Java Application**.



```
import java.util.HashMap;

/**
 *  Auth:
 *  This interface is used to authenticate third-party systems before third-party systems access open APIs.
 */
ublic class Authentication {
    @SuppressWarnings("unchecked")
    public static void main(String args[]) throws Exception {

        // Two-Way Authentication
        HttpsUtil httpsUtil = new HttpsUtil();
        httpsUtil.initSSLConfigForTwoWay();

        String appId = Constant.APPID;
        String secret = Constant.SECRET;
        String urlLogin = Constant.APP_AUTH;

        Map<String, String> param = new HashMap<>();
        param.put("appId", appId);
        param.put("secret", secret);

        StreamClosedHttpResponse responseLogin = httpsUtil.doPostFormUrlEncodedGetStatusLine(urlLogin, param);

        System.out.println("app auth success,return accessToken:");
        System.out.println(responseLogin.getStatusLine());
        System.out.println(responseLogin.getContent());
        System.out.println();

        //resolve the value of accessToken from responseLogin.
        Map<String, String> data = new HashMap<>();
        data = JsonUtil.jsonString2SimpleObj(responseLogin.getContent(), data.getClass());
        String accessToken = data.get("accessToken");
        System.out.println("accessToken:" + accessToken);

    }
```

Step 4 View the response log on the console. If an access token is obtained, the authentication is successful.

Keep the access token securely. It will be used when other APIs are called.



```
<terminated> Authentication [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (2018年3月14日 下午4:37:54)
app auth success,return accessToken:
HTTP/1.1 200 OK{"accessToken":"8d44bc6cdb60e863dfdbb8ccfed4e51","tokenType":"bearer","refreshToken":"9e11397df47877fe2afe4cd5c463621","expiresIn":3600,

accessToken:8d44bc6cdb60e863dfdbb8ccfed4e51
```

NOTE

- If no correct response is obtained, check whether the global constants are modified incorrectly or a network fault occurs. You can locate the fault by following the instructions provided in **Performing Single-Step Debugging**.
- An access token expires after the period specified by **expiresIn** elapses. The unit of **expiresIn** is **seconds**.
- If an access token expires, you must obtain a new one. You can use the authentication API or the refresh token obtained during the previous authentication to obtain another access token. For details about the refresh token, see **RefreshToken.java** in the code sample and the API reference document.
- **Northbound JAVA API Demo** provides examples of messages for calling each API. For details, see **src** > **resource** > **demo_TCP_message.json**.

----End

1.5.2 Data Subscription

Overview

An NA calls the **Subscribing to Service Data of the IoT Platform** API to notify the IoT platform of message push addresses and notification types, such as device service data and device alarms. For details about the subscription API, see the API reference document.

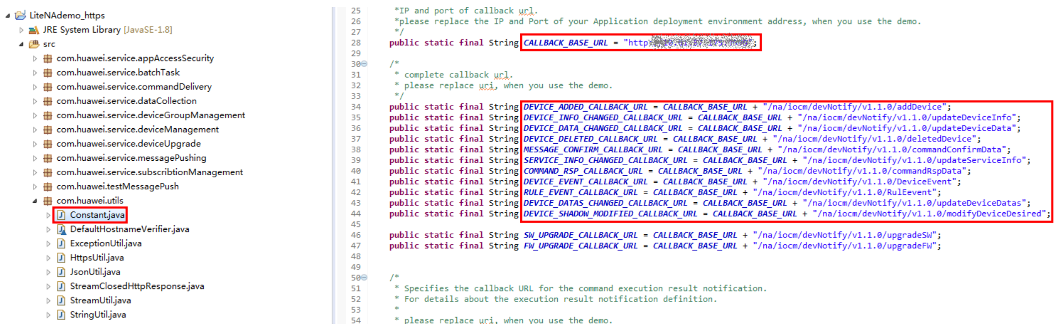
In the subscription scenario, the IoT platform is the client, and the NA is the server. The IoT platform calls the API of the NA and pushes messages to the NA. If the subscription callback URL is an HTTPS address, a CA certificate must be uploaded to the IoT platform. The CA certificate is provided by the NA. (For details on how to obtain the certificate, see **Exporting a CA Certificate**.) To load a CA certificate, choose **Applications** > **Interconnection**, and click **Certificate Management** and **Add** in the **Push Certificate** area. For details, see **Uploading a CA Certificate**.

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

Procedure

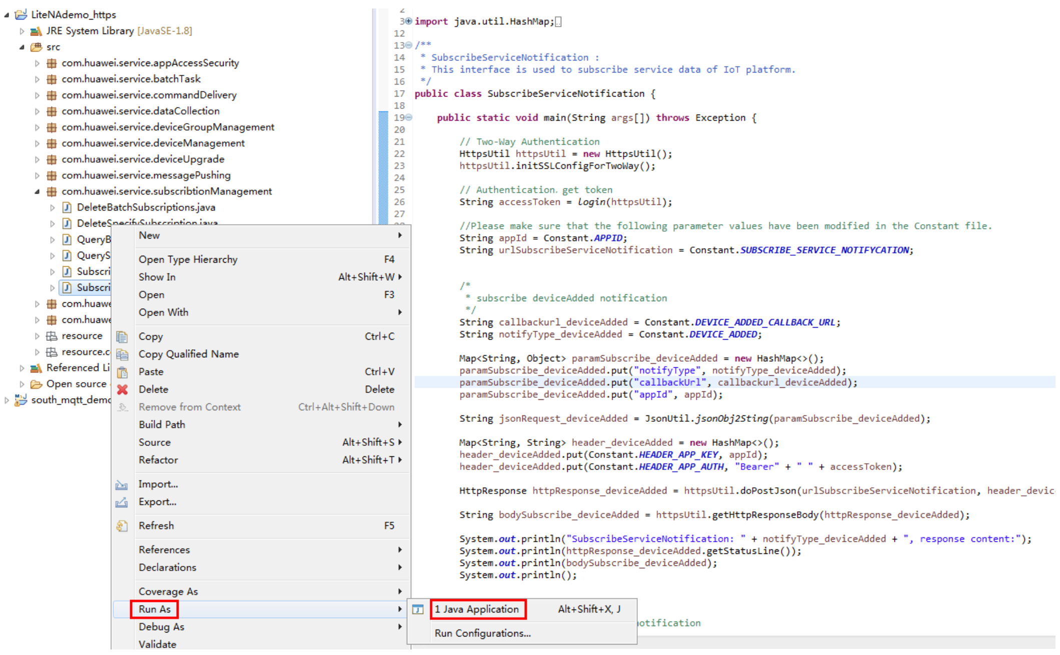
Step 1 In the Eclipse, choose **src** > **com.huawei.utils** > **Constant.java**, modify **CALLBACK_BASE_URL**, and enter the callback URL and port number.

In the same application, the callback URL and port number of all subscription types must be the same. **Subscription Test** of the Developer Center checks the validity and connectivity of callback URLs.


LiteNAdemo_https
JRE System Library [JavaSE-1.8]
src
com.huawei.service.appAccessSecurity
com.huawei.service.batchTask
com.huawei.service.commandDelivery
com.huawei.service.dataCollection
com.huawei.service.deviceGroupManagement
com.huawei.service.deviceManagement
com.huawei.service.deviceUpgrade
com.huawei.service.messagePushing
com.huawei.service.subscribtionManagement
com.huawei.testMessagePush
com.huawei.utils
Constant.java
DefaultHostnameVerifier.java
ExceptionUtil.java
HttpsUtil.java
JsonUtil.java
StreamClosedHttpResponse.java
StreamUtil.java
StringUtil.java

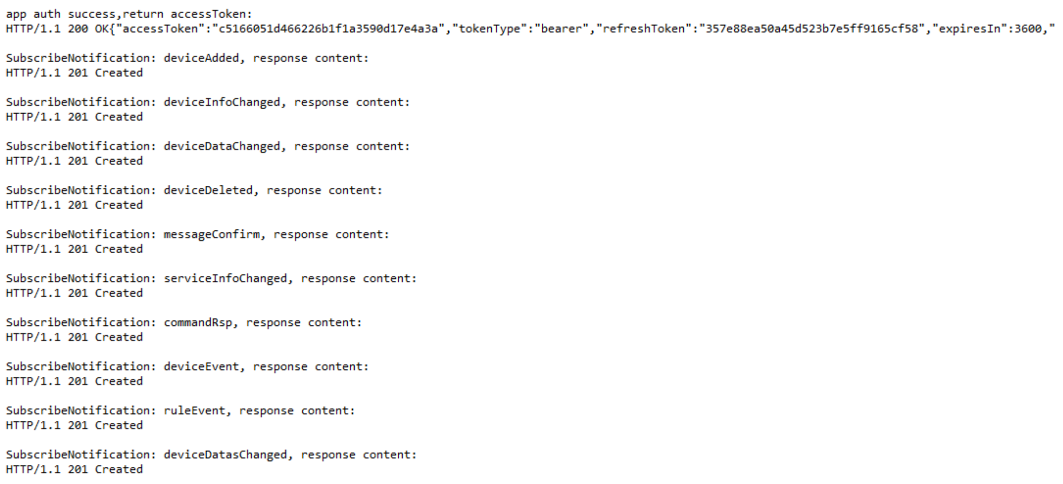
```
     *IP and port of callback url.
     *please replace the IP and Port of your Application deployment environment address, when you use the demo.
     */
    public static final String CALLBACK_BASE_URL = "http[redacted]";

    /*
     * complete callback url.
     * please replace uri, when you use the demo.
     */
    public static final String DEVICE_ADDED_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/addDevice";
    public static final String DEVICE_INFO_CHANGED_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/updateDeviceInfo";
    public static final String DEVICE_DATA_CHANGED_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/updateDeviceData";
    public static final String DEVICE_DELETED_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/deletedDevice";
    public static final String MESSAGE_CONFIRM_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/commandConfirmData";
    public static final String SERVICE_INFO_CHANGED_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/updateServiceInfo";
    public static final String COMMAND_RSP_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/commandRspData";
    public static final String DEVICE_EVENT_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/DeviceEvent";
    public static final String RULE_EVENT_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/RulEevent";
    public static final String DEVICE_DATAS_CHANGED_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/updateDeviceDatas";
    public static final String DEVICE_SHADOW_MODIFIED_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/modifyDeviceDesired";

    public static final String SW_UPGRADE_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/upgradeSW";
    public static final String FW_UPGRADE_CALLBACK_URL = CALLBACK_BASE_URL + "/na/iocm/devNotify/v1.1.0/upgradeFW";


    /*
     * Specifies the callback URL for the command execution result notification.
     * For details about the execution result notification definition.
     *
     * please replace uri, when you use the demo.
```

Step 2 In the Eclipse, choose **src** > **com.huawei.service.subscribtionManagement**, right-click **SubscribeServiceNotification.java**, and choose **Run As** > **Java Application**.


LiteNAdemo_https
JRE System Library [JavaSE-1.8]
src
com.huawei.service.appAccessSecurity
com.huawei.service.batchTask
com.huawei.service.commandDelivery
com.huawei.service.dataCollection
com.huawei.service.deviceGroupManagement
com.huawei.service.deviceManagement
com.huawei.service.deviceUpgrade
com.huawei.service.messagePushing
com.huawei.service.subscribtionManagement
DeleteBatchSubscriptions.java
DeleteSpecifySubscription.java
QueryB
QueryS
Subscri
Subscri
com.huaw
com.huaw
resource
resource.c
Referenced Li
Open source
south_mqtt_demc
New
Open Type Hierarchy F4
Show In Alt+Shift+W
Open F3
Open With
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Remove from Context Ctrl+Alt+Shift+Down
Build Path
Source Alt+Shift+S
Refactor Alt+Shift+T
Import...
Export...
Refresh F5
References
Declarations
Coverage As
Run As
Debug As
Validate
1 Java Application Alt+Shift+X, J
Run Configurations...

```
import java.util.HashMap;

/**
 * SubscribeServiceNotification :
 * This interface is used to subscribe service data of IoT platform.
 */
public class SubscribeServiceNotification {

    public static void main(String args[]) throws Exception {

        // Two-Way Authentication
        HttpsUtil httpsUtil = new HttpsUtil();
        httpsUtil.initSSLConfigForTwoWay();

        // Authentication. get token
        String accessToken = login(httpsUtil);

        //Please make sure that the following parameter values have been modified in the Constant file.
        String appId = Constant.APPID;
        String urlSubscribeServiceNotification = Constant.SUBSCRIBE_SERVICE_NOTIFYCATION;


        /*
         * subscribe deviceAdded notification
         */
        String callbackurl_deviceAdded = Constant.DEVICE_ADDED_CALLBACK_URL;
        String notifyType_deviceAdded = Constant.DEVICE_ADDED;

        Map<String, Object> paramSubscribe_deviceAdded = new HashMap<>();
        paramSubscribe_deviceAdded.put("notifyType", notifyType_deviceAdded);
        paramSubscribe_deviceAdded.put("callbackUrl", callbackurl_deviceAdded);
        paramSubscribe_deviceAdded.put("appId", appId);

        String jsonRequest_deviceAdded = JsonUtil.jsonObj2Sting(paramSubscribe_deviceAdded);

        Map<String, String> header_deviceAdded = new HashMap<>();
        header_deviceAdded.put(Constant.HEADER_APP_KEY, appId);
        header_deviceAdded.put(Constant.HEADER_APP_AUTH, "Bearer" + " " + accessToken);

        HttpResponse httpResponse_deviceAdded = httpsUtil.doPostJson(urlSubscribeServiceNotification, header_devic

        String bodySubscribe_deviceAdded = httpsUtil.getHttpResponseBody(httpResponse_deviceAdded);

        System.out.println("SubscribeServiceNotification: " + notifyType_deviceAdded + ", response content:");
        System.out.println(httpResponse_deviceAdded.getStatusLine());
        System.out.println(bodySubscribe_deviceAdded);
        System.out.println();


                                                otification
```

Step 3 View the response log on the console. If all types of subscriptions receive "201 Created" response, the subscription is successful.



```
app auth success,return accessToken:
HTTP/1.1 200 OK{"accessToken":"c5166051d466226b1f1a3590d17e4a3a","tokenType":"bearer","refreshToken":"357e88ea50a45d523b7e5ff9165cf58","expiresIn":3600,"

SubscribeNotification: deviceAdded, response content:
HTTP/1.1 201 Created

SubscribeNotification: deviceInfoChanged, response content:
HTTP/1.1 201 Created

SubscribeNotification: deviceDataChanged, response content:
HTTP/1.1 201 Created

SubscribeNotification: deviceDeleted, response content:
HTTP/1.1 201 Created

SubscribeNotification: messageConfirm, response content:
HTTP/1.1 201 Created

SubscribeNotification: serviceInfoChanged, response content:
HTTP/1.1 201 Created

SubscribeNotification: commandRsp, response content:
HTTP/1.1 201 Created

SubscribeNotification: deviceEvent, response content:
HTTP/1.1 201 Created

SubscribeNotification: ruleEvent, response content:
HTTP/1.1 201 Created

SubscribeNotification: deviceDatasChanged, response content:
HTTP/1.1 201 Created
```

NOTE

- To modify the callback URL, change the value of **CALLBACK_BASE_URL** in the **Constants.java** file and run

  SubscribeServiceNotification.java. The new callback URL replaces the original one.
- After the subscription is complete, you can choose **src** > **com.huawei.testMessagePush** > **SimpleHttpServer.java** to set up an NA to receive messages (for example, POST messages) pushed by the IoT platform. If you need to perform a local test on the callback function and view the callback content, use the class **src** > **com.huawei.testMessagePush** > **TestSubscribeAllServiceNotification.java** provided in the **Northbound JAVA API Demo** and refer to the operations in **Data Reporting**.

----End

1.5.3 Device Registration

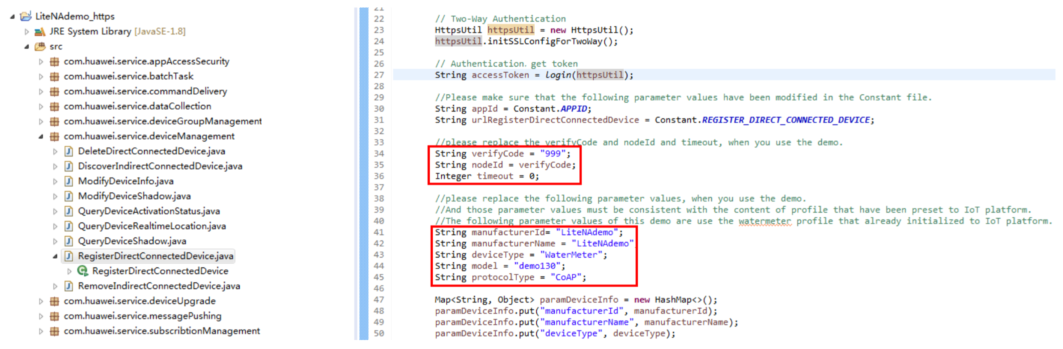
Overview

The NA server calls the API for registering a directly connected device to add devices to the IoT platform. For details about the API, see the API reference document.

This topic describes how to call the API for registering a directly connected device based on the Java code sample of the API.

Procedure

Step 1 In the Eclipse, choose **src** > **com.huawei.service.deviceManagement** > **RegisterDirectConnectedDevice.java**, and change the values of **verifyCode**, **nodeId**, **timeout**, **manufacturerId**, **manufacturerName**, **deviceType**, **model**, and **protocolType**.



```
LiteNAdemo_https
  JRE System Library [JavaSE-1.8]
  src
    com.huawei.service.appAccessSecurity
    com.huawei.service.batchTask
    com.huawei.service.commandDelivery
    com.huawei.service.dataCollection
    com.huawei.service.deviceGroupManagement
    com.huawei.service.deviceManagement
      DeleteDirectConnectedDevice.java
      DiscoverIndirectConnectedDevice.java
      ModifyDeviceInfo.java
      ModifyDeviceShadow.java
      QueryDeviceActivationStatus.java
      QueryDeviceRealtimeLocation.java
      QueryDeviceShadow.java
      RegisterDirectConnectedDevice.java
        RegisterDirectConnectedDevice
      RemoveIndirectConnectedDevice.java
    com.huawei.service.deviceUpgrade
    com.huawei.service.messagePushing
    com.huawei.service.subscribtionManagement
```

```
        // Two-Way Authentication
        HttpsUtil httpsUtil = new HttpsUtil();
        httpsUtil.initSSLConfigForTwoWay();

        // Authentication. get token
        String accessToken = login(httpsUtil);

        //Please make sure that the following parameter values have been modified in the Constant file.
        String appId = Constant.APPID;
        String urlRegisterDirectConnectedDevice = Constant.REGISTER_DIRECT_CONNECTED_DEVICE;

        //please replace the verifyCode and nodeId and timeout, when you use the demo.
        String verifyCode = "999";
        String nodeId = verifyCode;
        Integer timeout = 0;

        //please replace the following parameter values, when you use the demo.
        //And those parameter values must be consistent with the content of profile that have been preset to IoT platform.
        //The following parameter values of this demo are use the watermeter profile that already initialized to IoT platform.
        String manufacturerId= "LiteNAdemo";
        String manufacturerName = "LiteNAdemo";
        String deviceType = "WaterMeter";
        String model = "demo130";
        String protocolType = "CoAP";

        Map<String, Object> paramDeviceInfo = new HashMap<>();
        paramDeviceInfo.put("manufacturerId", manufacturerId);
        paramDeviceInfo.put("manufacturerName", manufacturerName);
        paramDeviceInfo.put("deviceType", deviceType);
```

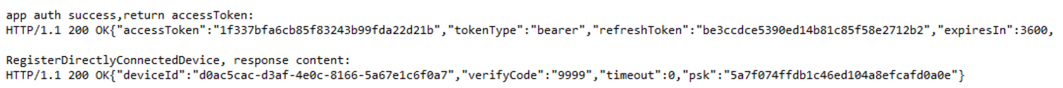
Parameters are described as follows:

- The values of **verifyCode** and **nodeId** must be the same as the IMEI or MAC address of a physical device. If a device simulator is used, the value of **verifyCode** can be a combination of digits, letters, and special characters. The value can be user-defined but must be unique.
- The unit of **timeout** is **second**. The values of **timeout** are as follows:
  - **0**: indicates that the device never expires.
  - **> 0**: indicates that the device must be made online within the specified period. Otherwise, the IoT platform removes the device immediately after the period expires. If **timeout** is not specified, the default interval (180 seconds) is used.
  - After a device is bound, **timeout** becomes invalid and the device will never expire.
- The values of **manufacturerId**, **manufacturerName**, **deviceType**, **model**, and **protocolType** must be the same as those in the profile file.

Step 2 Right-click **RegisterDirectConnectedDevice.java** and choose **Run As** > **Java Application**.

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

Developers can check whether the newly registered device is displayed on the **Product** > **Device Management** of the Developer Center. In this case, the registered device has only the device ID information.



```
app auth success,return accessToken:
HTTP/1.1 200 OK{"accessToken":"1f337bfa6cb85f83243b99fda22d21b","tokenType":"bearer","refreshToken":"be3ccdce5390ed14b81c85f58e2712b2","expiresIn":3600,

RegisterDirectlyConnectedDevice, response content:
HTTP/1.1 200 OK{"deviceId":"d0ac5cac-d3af-4e0c-8166-5a67e1c6f0a7","verifyCode":"9999","timeout":0,"psk":"5a7f074ffdb1c46ed104a8efcafd0a0e"}
```

----End

1.5.4 Device Access to the IoT Platform

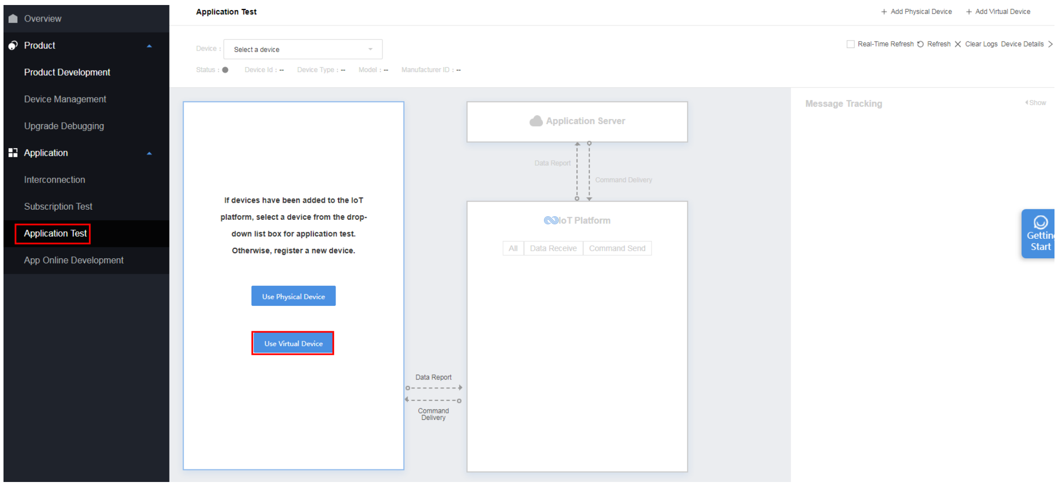
Overview

After devices are connected to the IoT platform, data can be exchanged between the IoT platform and NA servers.

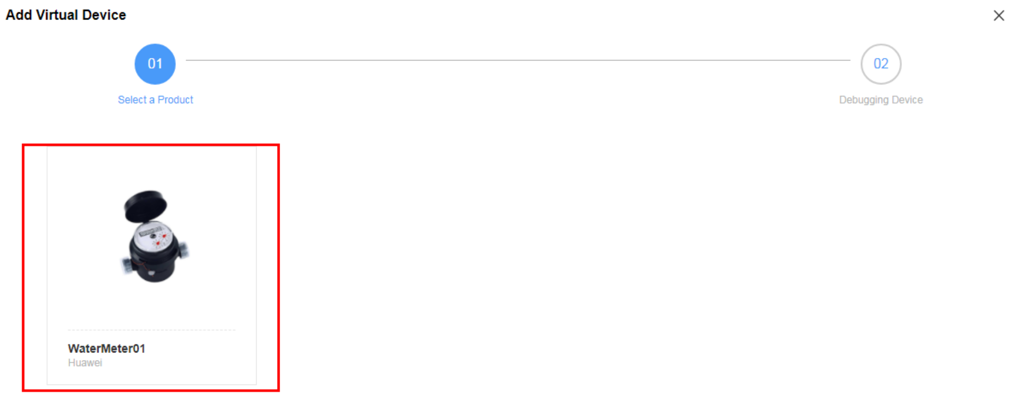
The Developer Center provides the application test function to simulate the scenario in which devices are connected to the IoT platform. Developers can also connect a physical device to the IoT platform to test the application. The following describes how to simulate device access to the IoT platform:

Procedure

Step 1 Choose **Applications** > **Application Test**. Click **Use Virtual Device**.



Step 2 In the **Add Virtual Device** dialog box displayed, select a device.



----End

1.5.5 Data Reporting

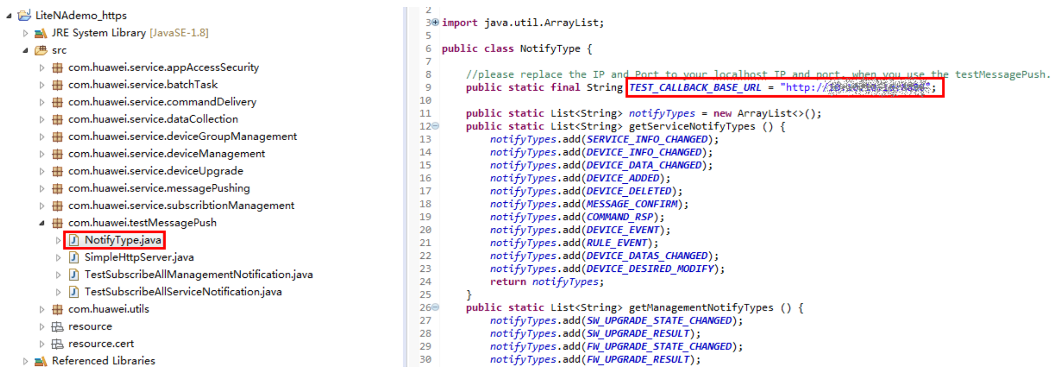
Overview

After a device reports data, the IoT platform pushes data reported by the device to the subscribed-to address. The Developer Center provides a device simulator to simulate the scenario where a real device reports data. Developers can also connect a physical device to report data.

This topic describes how to report data using the device simulator based on the Java code sample of the data reporting API. A simple HTTP server is provided in the Java code sample of the API to help developers test whether the IoT platform has pushed messages to the subscribed-to address.

Procedure

Step 1 In the Eclipse, choose **src** > **com.huawei.testMessagePush** > **NotifyType.java**. Modify the value of **TEST_CALLBACK_BASE_URL**, and enter the local IP address and port number. The port number cannot be used by other local programs.



```
LiteNAdemo_https
  JRE System Library [JavaSE-1.8]
  src
    com.huawei.service.appAccessSecurity
    com.huawei.service.batchTask
    com.huawei.service.commandDelivery
    com.huawei.service.dataCollection
    com.huawei.service.deviceGroupManagement
    com.huawei.service.deviceManagement
    com.huawei.service.deviceUpgrade
    com.huawei.service.messagePushing
    com.huawei.service.subscribtionManagement
    com.huawei.testMessagePush
      NotifyType.java
      SimpleHttpServer.java
      TestSubscribeAllManagementNotification.java
      TestSubscribeAllServiceNotification.java
    com.huawei.utils
    resource
    resource.cert
  Referenced Libraries
```

```java

import java.util.ArrayList;

public class NotifyType {

    //please replace the IP and Port to your localhost IP and port, when you use the testMessagePush.
    public static final String TEST_CALLBACK_BASE_URL = "http://[redacted]";

    public static List<String> notifyTypes = new ArrayList<>();
    public static List<String> getServiceNotifyTypes () {
        notifyTypes.add(SERVICE_INFO_CHANGED);
        notifyTypes.add(DEVICE_INFO_CHANGED);
        notifyTypes.add(DEVICE_DATA_CHANGED);
        notifyTypes.add(DEVICE_ADDED);
        notifyTypes.add(DEVICE_DELETED);
        notifyTypes.add(MESSAGE_CONFIRM);
        notifyTypes.add(COMMAND_RSP);
        notifyTypes.add(DEVICE_EVENT);
        notifyTypes.add(RULE_EVENT);
        notifyTypes.add(DEVICE_DATAS_CHANGED);
        notifyTypes.add(DEVICE_DESIRED_MODIFY);
        return notifyTypes;
    }
    public static List<String> getManagementNotifyTypes () {
        notifyTypes.add(SW_UPGRADE_STATE_CHANGED);
        notifyTypes.add(SW_UPGRADE_RESULT);
        notifyTypes.add(FW_UPGRADE_STATE_CHANGED);
        notifyTypes.add(FW_UPGRADE_RESULT);
```

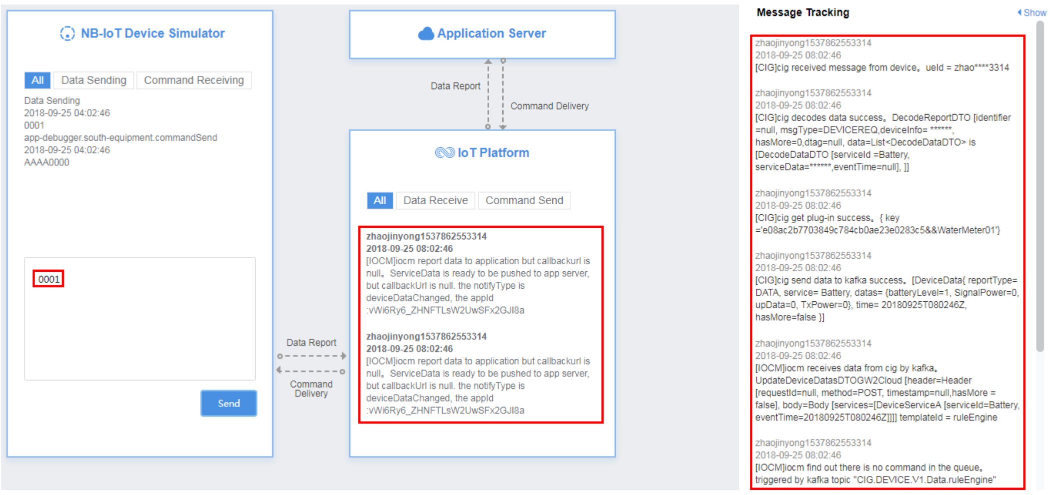
Step 2 Right-click **src** > **com.huawei.testMessagePush** > **TestSubscribeAllServiceNotification.java**, and choose **Run As** > **Java Application**.

Step 3 In the project space, choose **Application** > **Application Test**. Use the virtual device added in **Device Access to the IoT Platform** to report data.

NOTE

Developers can also connect a physical device to report data.

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



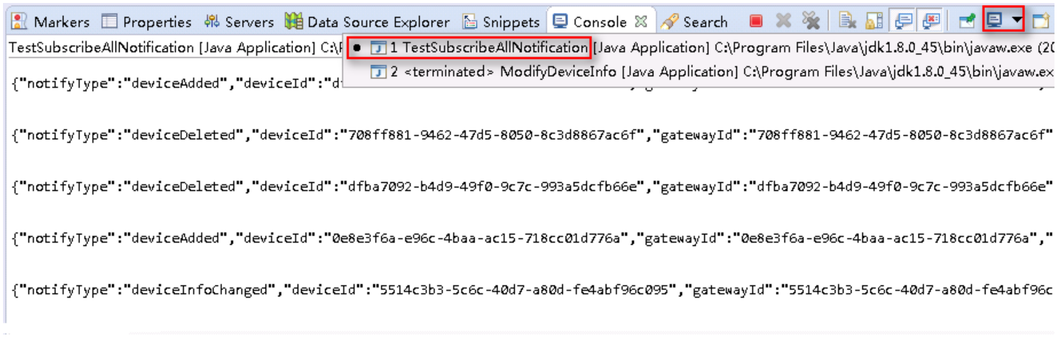
```
Data Sending
2018-09-25 04:02:46
0001
app-debugger.south-equipment.commandSend
2018-09-25 04:02:46
AAAA0000
```

```
zhaojinyong1537862553314
2018-09-25 08:02:46
[IOCM]iocm report data to application but callbackurl is null。ServiceData is ready to be pushed to app server, but callbackUrl is null. the notifyType is deviceDataChanged, the appId :vWi6Ry6_ZHNFTLsW2UwSFx2GJl8a

zhaojinyong1537862553314
2018-09-25 08:02:46
[IOCM]iocm report data to application but callbackurl is null。ServiceData is ready to be pushed to app server, but callbackUrl is null. the notifyType is deviceDataChanged, the appId :vWi6Ry6_ZHNFTLsW2UwSFx2GJl8a
```

```
zhaojinyong1537862553314
2018-09-25 08:02:46
[CIG]cig received message from device。ueId = zhao****3314

zhaojinyong1537862553314
2018-09-25 08:02:46
[CIG]cig decodes data success。DecodeReportDTO [identifier =null, msgType=DEVICEREQ,deviceInfo= ******, hasMore=0,dtag=null, data=List<DecodeDataDTO> is [DecodeDataDTO [serviceId =Battery, serviceData=******,eventTime=null], ]]

zhaojinyong1537862553314
2018-09-25 08:02:46
[CIG]cig get plug-in success。{ key ='e08ac2b7703849c784cb0ae23e0283c5&&WaterMeter01'}

zhaojinyong1537862553314
2018-09-25 08:02:46
[CIG]cig send data to kafka success。[DeviceData{ reportType= DATA, service= Battery, datas= {batteryLevel=1, SignalPower=0, upData=0, TxPower=0}, time= 20180925T080246Z, hasMore=false }]

zhaojinyong1537862553314
2018-09-25 08:02:46
[IOCM]iocm receives data from cig by kafka。UpdateDeviceDatasDTOGW2Cloud [header=Header [requestId=null, method=POST, timestamp=null,hasMore = false], body=Body [services=[DeviceServiceA [serviceId=Battery, eventTime=20180925T080246Z]]]] templateId = ruleEngine

zhaojinyong1537862553314
2018-09-25 08:02:46
[IOCM]iocm find out there is no command in the queue。triggered by kafka topic "CIG.DEVICE.V1.Data.ruleEngine"
```

Step 4 In the Eclipse, choose the **TestSubscribeAllNotification.java** console and view the messages pushed by the IoT platform to the NA server.

Developers can also test the subscription result. For example, if **deviceAdded** is subscribed to, developers can view messages pushed by the IoT platform on the **TestSubscribeAllNotification.java** console after performing the operations described in **Device Registration**.



```
TestSubscribeAllNotification [Java Application] C:\I
● 1 TestSubscribeAllNotification [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (20
  2 <terminated> ModifyDeviceInfo [Java Application] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.ex
{"notifyType":"deviceAdded","deviceId":"d

{"notifyType":"deviceDeleted","deviceId":"708ff881-9462-47d5-8050-8c3d8867ac6f","gatewayId":"708ff881-9462-47d5-8050-8c3d8867ac6f"

{"notifyType":"deviceDeleted","deviceId":"dfba7092-b4d9-49f0-9c7c-993a5dcfb66e","gatewayId":"dfba7092-b4d9-49f0-9c7c-993a5dcfb66e"

{"notifyType":"deviceAdded","deviceId":"0e8e3f6a-e96c-4baa-ac15-718cc01d776a","gatewayId":"0e8e3f6a-e96c-4baa-ac15-718cc01d776a","

{"notifyType":"deviceInfoChanged","deviceId":"5514c3b3-5c6c-40d7-a80d-fe4abf96c095","gatewayId":"5514c3b3-5c6c-40d7-a80d-fe4abf96c
```

----End

1.5.6 Command Delivery

Overview

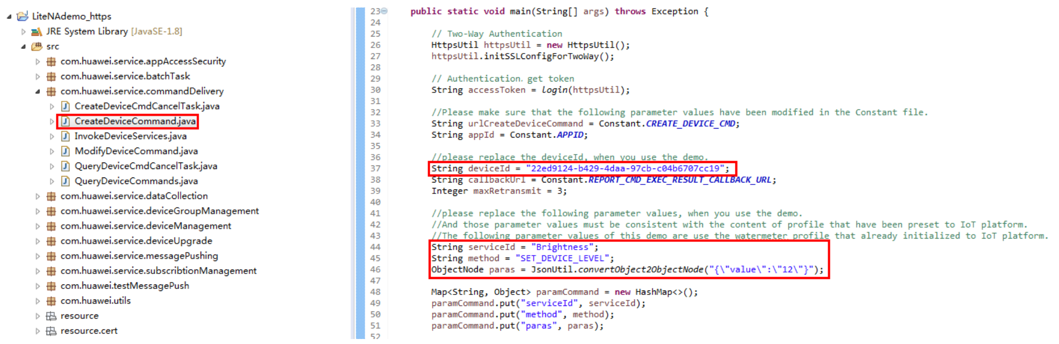
The NA server calls the device command creation API or device service calling API of the IoT platform to deliver control instructions to devices. For details about the APIs, see the API reference document.

- When the access protocol at the application layer is LWM2M, the **Creating Device Commands** API is called to deliver commands.
- When the access protocol at the application layer is MQTT, the **Calling Device Services** API is called to deliver commands.

This topic describes how to deliver commands with the **Calling Device Services** API and the Java code sample of the API.

Procedure

Step 1 In the Eclipse, choose **src** > **com.huawei.service.commandDelivery** > **CreateDeviceCommand.java**, and change the values of **deviceId**, **serviceId**, **method**, and **paras**.



```
LiteNAdemo_https
  JRE System Library [JavaSE-1.8]
  src
    com.huawei.service.appAccessSecurity
    com.huawei.service.batchTask
    com.huawei.service.commandDelivery
      CreateDeviceCmdCancelTask.java
      CreateDeviceCommand.java
      InvokeDeviceServices.java
      ModifyDeviceCommand.java
      QueryDeviceCmdCancelTask.java
      QueryDeviceCommands.java
    com.huawei.service.dataCollection
    com.huawei.service.deviceGroupManagement
    com.huawei.service.deviceManagement
    com.huawei.service.deviceUpgrade
    com.huawei.service.messagePushing
    com.huawei.service.subscribtionManagement
    com.huawei.testMessagePush
    com.huawei.utils
    resource
    resource.cert
```

```java
public static void main(String[] args) throws Exception {

    // Two-Way Authentication
    HttpsUtil httpsUtil = new HttpsUtil();
    httpsUtil.initSSLConfigForTwoWay();

    // Authentication. get token
    String accessToken = login(httpsUtil);

    //Please make sure that the following parameter values have been modified in the Constant file.
    String urlCreateDeviceCommand = Constant.CREATE_DEVICE_CMD;
    String appId = Constant.APPID;

    //please replace the deviceId, when you use the demo.
    String deviceId = "22ed9124-b429-4daa-97cb-c04b6707cc19";
    String callbackUrl = Constant.REPORT_CMD_EXEC_RESULT_CALLBACK_URL;
    Integer maxRetransmit = 3;

    //please replace the following parameter values, when you use the demo.
    //And those parameter values must be consistent with the content of profile that have been preset to IoT platform.
    //The following parameter values of this demo are use the watermeter profile that already initialized to IoT platform.
    String serviceId = "Brightness";
    String method = "SET_DEVICE_LEVEL";
    ObjectNode paras = JsonUtil.convertObject2ObjectNode("{\"value\":\"12\"}");

    Map<String, Object> paramCommand = new HashMap<>();
    paramCommand.put("serviceId", serviceId);
    paramCommand.put("method", method);
    paramCommand.put("paras", paras);
```

Parameters are described as follows:

- The value of **deviceId** is obtained when a device is registered.
- The values of **serviceId**, **method**, and **paras** are the same as those defined in the profile file.

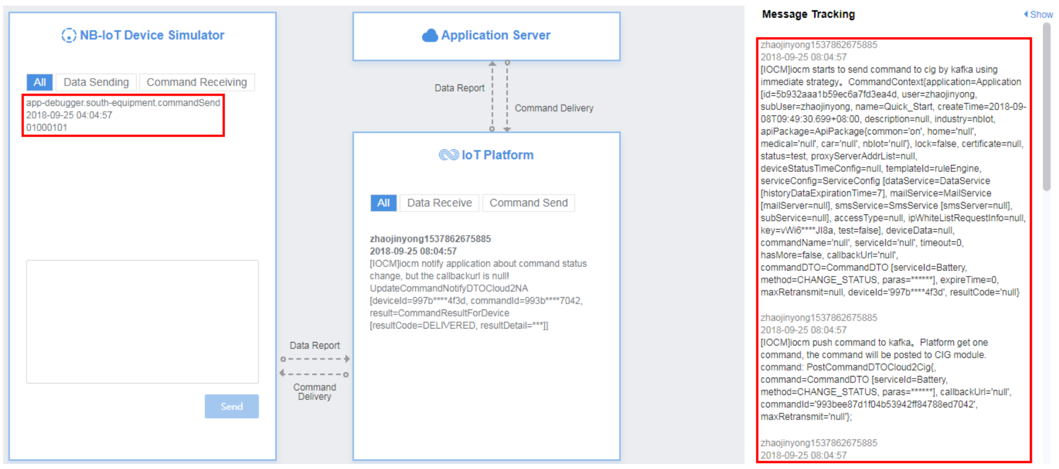
Step 2 Right-click **CreateDeviceCommand.java** and choose **Run As** > **Java Application**.

Step 3 View the command delivery log on the console. If the **201 Created** response is received, the command is delivered to the IoT platform.

```
app auth success,return accessToken:
HTTP/1.1 200 OK{"accessToken":"49b9b60d439d9f018a23c2c25d9e71","tokenType":"bearer","refreshToken":"51762486bb4df2142f904696a8ffb743","expiresIn":3600,"scope

PostAsynCommand, response content:
HTTP/1.1 201 Created{"commandId":"7e7ed2fd17c2449582c71adb0efa18b5","appId":"pQJNoCHou8mBo7anA7Fkw07luOAa","deviceId":"85ef387c-49cd-455d-9f70-a1bf1a990054",
```

If the application test function of the Developer Center is used to simulate device access and data reporting, developers can select the virtual device created in **Device Access to the IoT Platform** to view the received commands by choosing **Application** > **Application Test**.

After the NA server delivers a command, view the received command (for example, a hexadecimal code stream) in **Device Simulator** and view processing logs of the IoT platform in **Message Tracking**.



```
app-debugger.south-equipment.commandSend
2018-09-25 04:04:57
01000101
```

```
zhaojinyong1537862675885
2018-09-25 08:04:57
[IOCM]iocm notify application about command status change, but the callbackurl is null! UpdateCommandNotifyDTOCloud2NA [deviceId=997b****4f3d, commandId=993b****7042, result=CommandResultForDevice [resultCode=DELIVERED, resultDetail=***]]
```

```
zhaojinyong1537862675885
2018-09-25 08:04:57
[IOCM]iocm starts to send command to cig by kafka using immediate strategy。CommandContext{application=Application [id=5b932aaa1b59ec6a7fd3ea4d, user=zhaojinyong, subUser=zhaojinyong, name=Quick_Start, createTime=2018-09-08T09:49:30.699+08:00, description=null, industry=nbIot, apiPackage=ApiPackage{common='on', home='null', medical='null', car='null', nbIot='null'}, lock=false, certificate=null, status=test, proxyServerAddrList=null, deviceStatusTimeConfig=null, templateId=ruleEngine, serviceConfig=ServiceConfig [dataService=DataService [historyDataExpirationTime=7], mailService=MailService [mailServer=null], smsService=SmsService [smsServer=null], subService=null], accessType=null, ipWhiteListRequestInfo=null, key=vWi6****JI8a, test=false], deviceData=null, commandName='null', serviceId='null', timeout=0, hasMore=false, callbackUrl='null', commandDTO=CommandDTO [serviceId=Battery, method=CHANGE_STATUS, paras=******], expireTime=0, maxRetransmit=null, deviceId='997b****4f3d', resultCode='null'}

zhaojinyong1537862675885
2018-09-25 08:04:57
[IOCM]iocm push command to kafka。Platform get one command, the command will be posted to CIG module. command: PostCommandDTOCloud2Cig{, command=CommandDTO [serviceId=Battery, method=CHANGE_STATUS, paras=******], callbackUrl='null', commandId='993bee87d1f04b53942ff84788ed7042', maxRetransmit='null'};

zhaojinyong1537862675885
2018-09-25 08:04:57
```

----End

1.5.7 Development of Other APIs

For details about how to develop other APIs, see the API reference document.

1.5.8 Reference

1.5.8.1 Preparing the Java Development Environment

This section uses Java as an example to describe the methods to install JDK, set environment variables, and install Eclipse.

1.5.8.1.1 Installing JDK 1.8

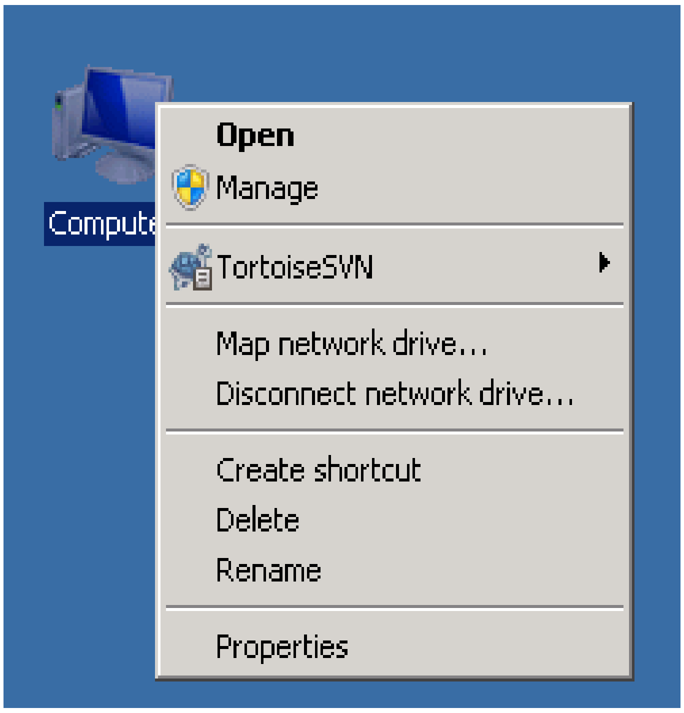
Download the JDK 1.8 installation package (for example, jdk-8u161-windows-x64.exe), and double-click it for installation.

The package is available at **http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html**.

1.5.8.1.2 Configuring Java Environment Variables (Windows OS)

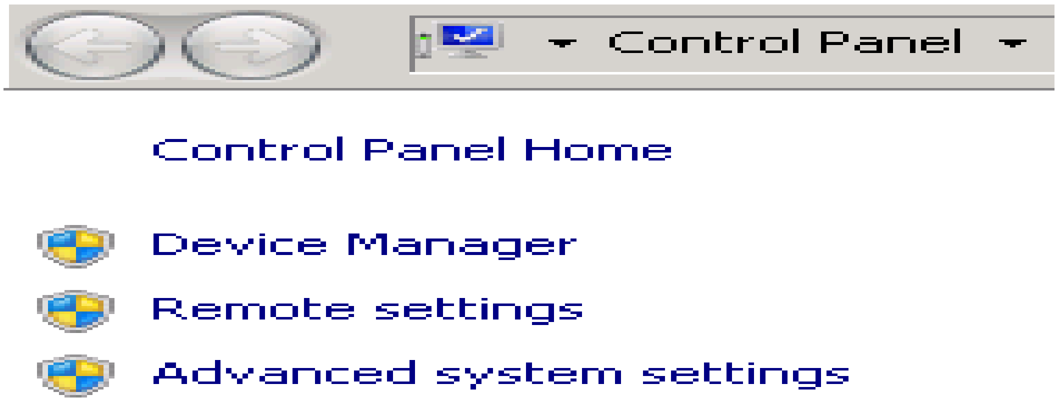
Step 1 Right-click **Computer** and choose **Properties**.

Figure 1-26 Properties



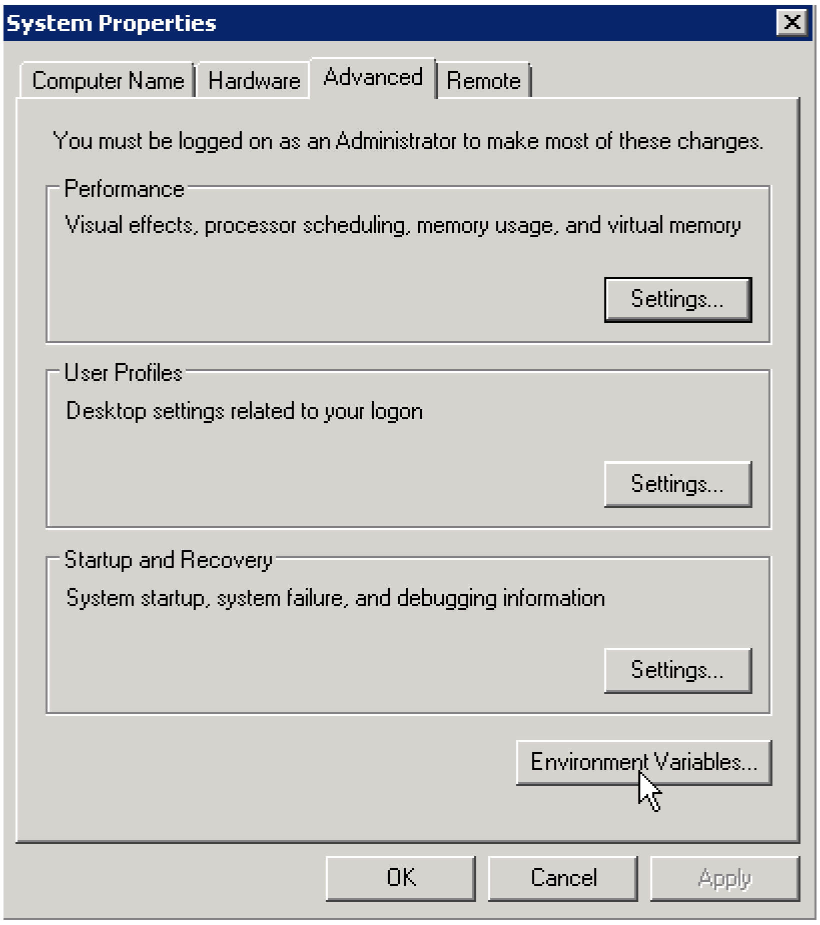
Step 2 Select **Advanced system settings**.

Figure 1-27 System



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

Figure 1-28 System Properties dialog box



You must be logged on as an Administrator to make most of these changes.

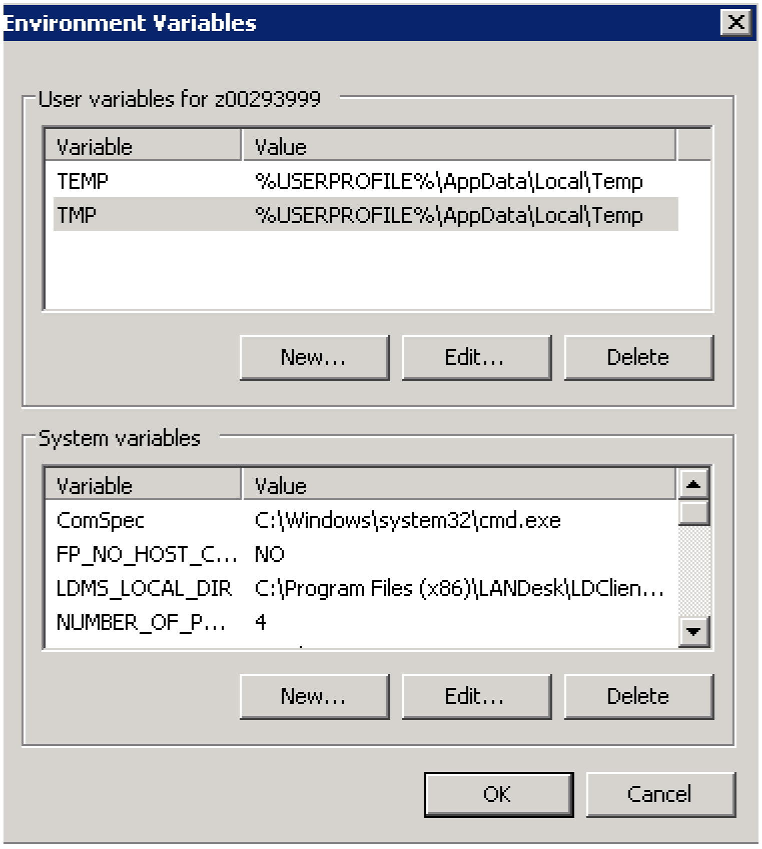
Visual effects, processor scheduling, memory usage, and virtual memory

Desktop settings related to your logon

System startup, system failure, and debugging information

Step 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 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.

Figure 1-29 Environment Variables dialog box

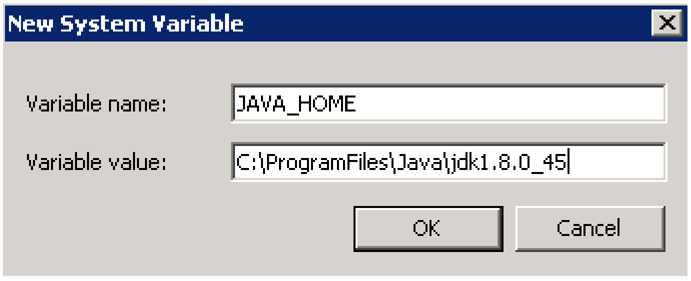


Variable	Value
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

Variable	Value
ComSpec	C:\Windows\system32\cmd.exe
FP_NO_HOST_C...	NO
LDMS_LOCAL_DIR	C:\Program Files (x86)\LANDesk\LDClien...
NUMBER_OF_P...	4

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.

Figure 1-30 Creating JAVA_HOME



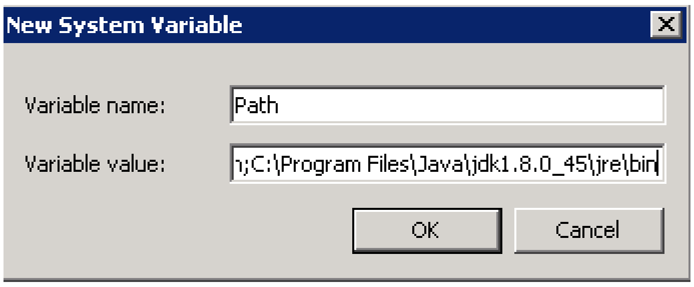
Variable name: JAVA_HOME

Variable value: C:\ProgramFiles\Java\jdk1.8.0_45

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**

Two paths need to be separated by using a semicolon (;).

Figure 1-31 Setting Path



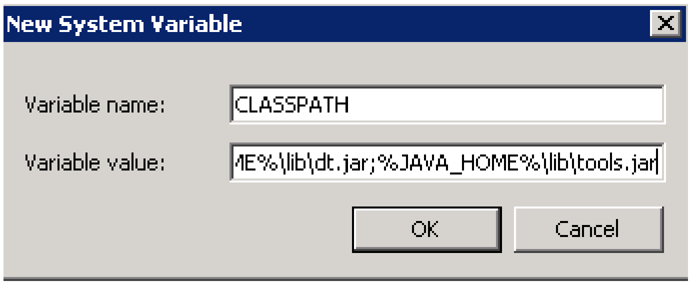
Variable name: Path
Variable value: ;C:\Program Files\Java\jdk1.8.0_45\jre\bin

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.

Figure 1-32 Setting CLASSPATH

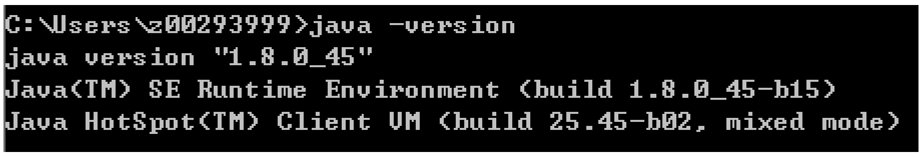


Variable name: CLASSPATH
Variable value: ME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar

Step 5 Restart the OS for the environment variables to take effect.

Step 6 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 successfully.

Figure 1-33 Verifying environment variables



```
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)
```

----End

1.5.8.1.3 Installing Eclipse

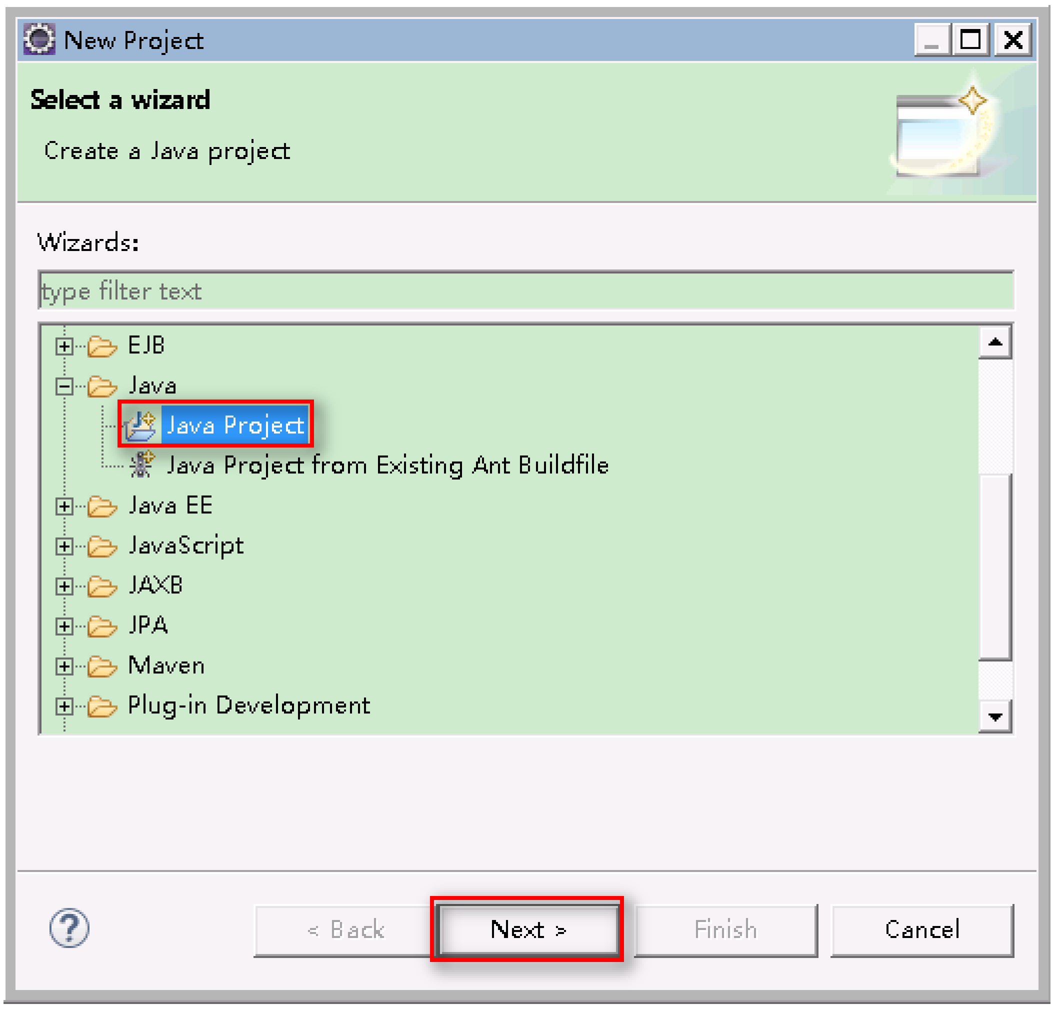
Download the Eclipse installation package and decompress it to a local directory. You can use the software without installation.

Eclipse is available on the official website at **http://www.eclipse.org/downloads**.

1.5.8.1.4 Creating a Project

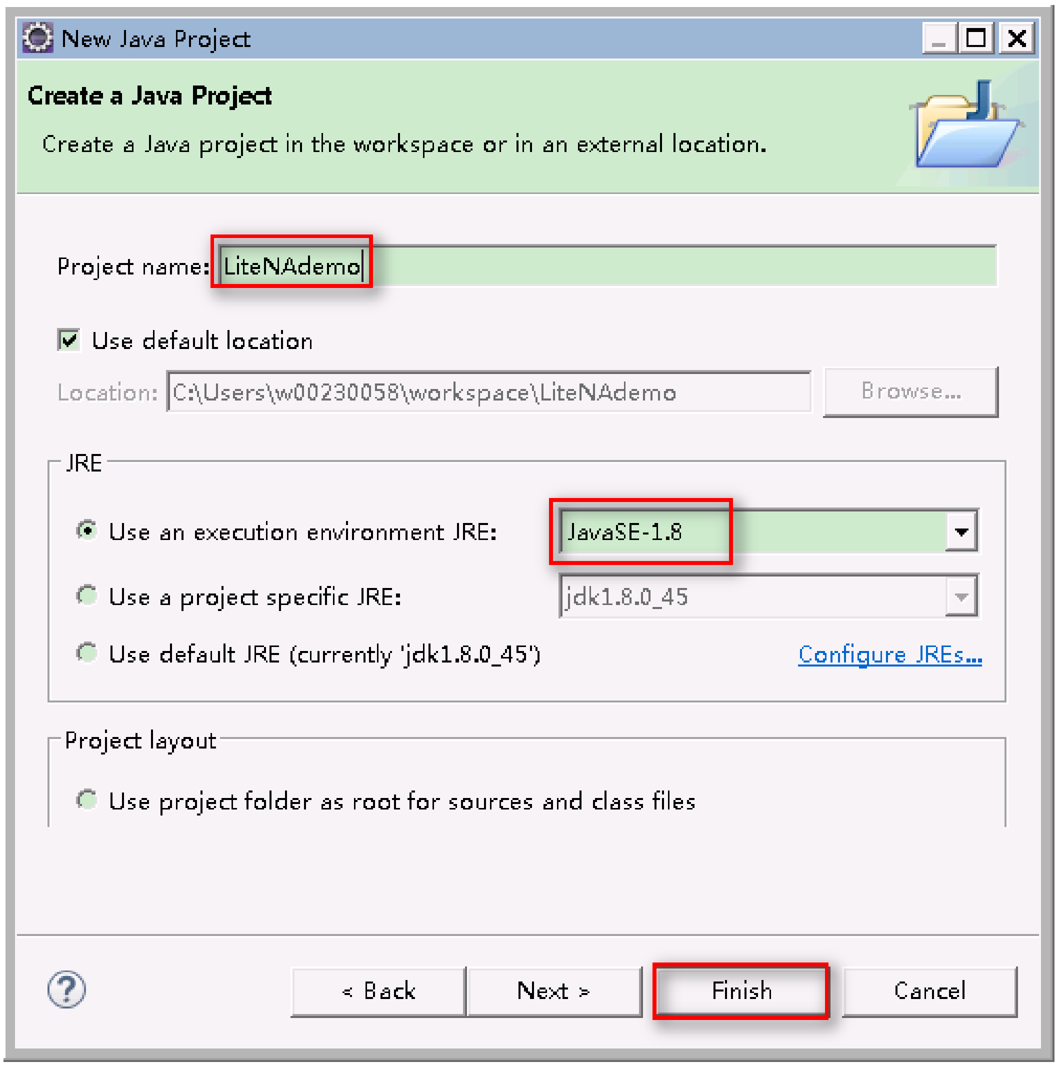
Step 1 In the Eclipse, and choose **File** > **New** > **Project**. In the dialog box displayed, select **Java Project** and click **Next**.

Figure 1-34 Creating a Java project



Step 2 Specify **Project name**, set the JRE version to **JavaSE-1.8**, and click **Finish**.

Figure 1-35 Setting the project name



New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: LiteNAdemo

[✓] Use default location

Location: C:\Users\w00230058\workspace\LiteNAdemo

JRE

(●) Use an execution environment JRE: JavaSE-1.8

( ) Use a project specific JRE: jdk1.8.0_45

( ) Use default JRE (currently 'jdk1.8.0_45') Configure JREs...

Project layout

( ) Use project folder as root for sources and class files

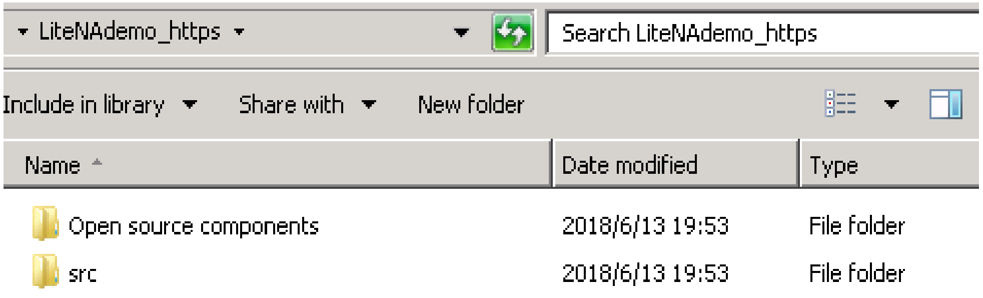
----End

1.5.8.1.5 Importing Code Example

Step 1 Decompress the API calling code example in Java (click **here** to obtain).

Step 2 After the decompression, copy the **Open source components** and **src** folders by pressing Ctrl +C.

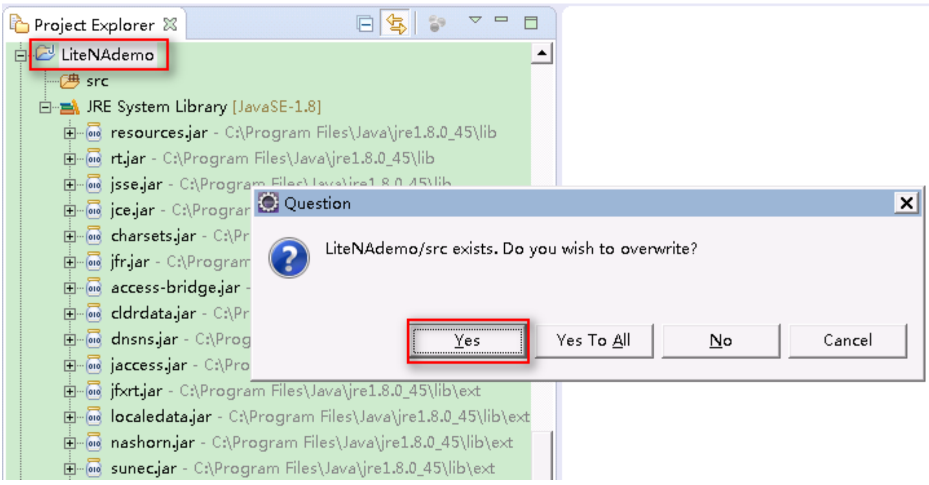
Figure 1-36 Copying the folders



Name	Date modified	Type
Open source components	2018/6/13 19:53	File folder
src	2018/6/13 19:53	File folder

Step 3 Open the project created in the Eclipse, select the project name, and paste the copied folders to the project directory.

Figure 1-37 Pasting the folders

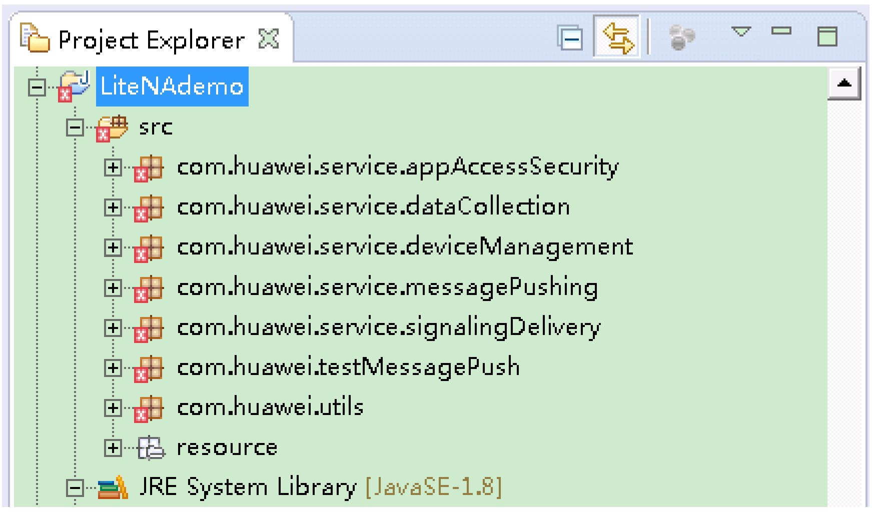


LiteNAdemo/src exists. Do you wish to overwrite?

Project Explorer
LiteNAdemo
src
JRE System Library [JavaSE-1.8]
resources.jar - C:\Program Files\Java\jre1.8.0_45\lib
rt.jar - C:\Program Files\Java\jre1.8.0_45\lib
jsse.jar
jce.jar
charsets.jar
jfr.jar
access-bridge.jar
cldrdata.jar
dnsns.jar
jaccess.jar
jfxrt.jar - C:\Program Files\Java\jre1.8.0_45\lib\ext
localedata.jar - C:\Program Files\Java\jre1.8.0_45\lib\ext
nashorn.jar - C:\Program Files\Java\jre1.8.0_45\lib\ext
sunec.jar - C:\Program Files\Java\jre1.8.0_45\lib\ext
Question
Yes
Yes To All
No
Cancel

After the paste is complete, files in the **src** directory are abnormal.

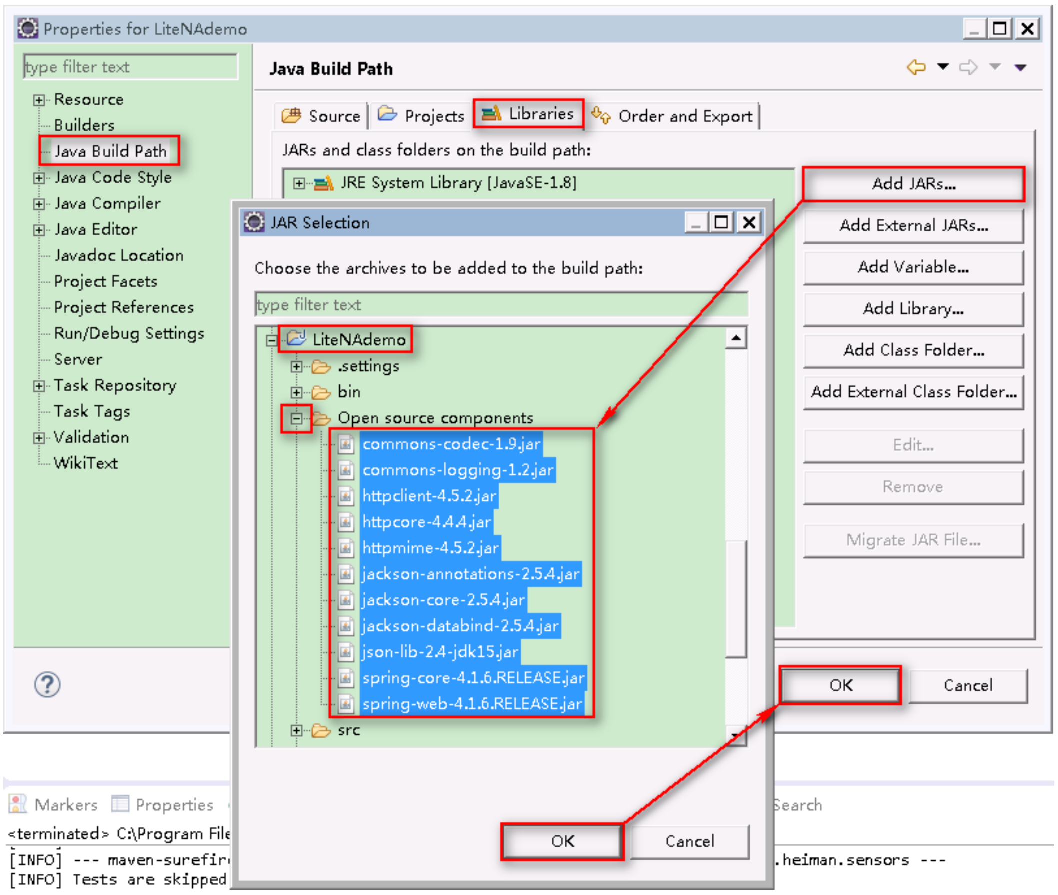
Figure 1-38 Abnormal files in the src directory



Project Explorer
LiteNAdemo
src
com.huawei.service.appAccessSecurity
com.huawei.service.dataCollection
com.huawei.service.deviceManagement
com.huawei.service.messagePushing
com.huawei.service.signalingDelivery
com.huawei.testMessagePush
com.huawei.utils
resource
JRE System Library [JavaSE-1.8]

Step 4 Right-click the project name and choose **Properties** > **Java Build Path** > **Libraries** > **Add JARs**. In the dialog box displayed, select all .jar files in the **Open source components** directory and click **OK**.

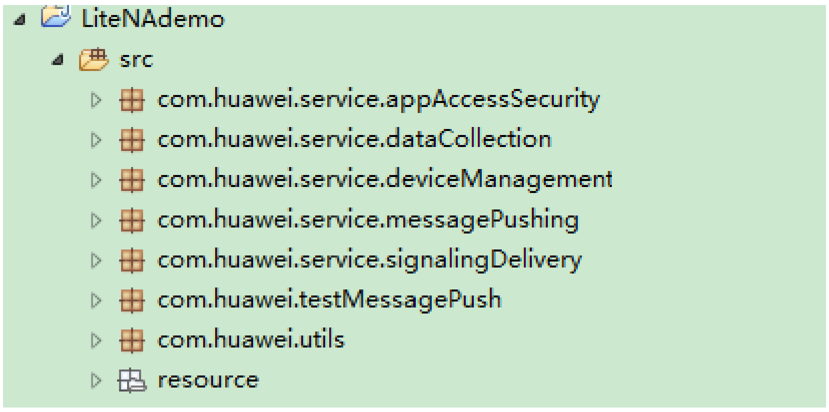
Figure 1-39 Importing .jar files



```
<terminated> C:\Program File
[INFO] --- maven-surefir                .heiman.sensors ---
[INFO] Tests are skipped
```

After the .jar files are imported, files in the **src** directory become normal.

Figure 1-40 Normal files in the src directory



```
LiteNAdemo
  src
    com.huawei.service.appAccessSecurity
    com.huawei.service.dataCollection
    com.huawei.service.deviceManagement
    com.huawei.service.messagePushing
    com.huawei.service.signalingDelivery
    com.huawei.testMessagePush
    com.huawei.utils
    resource
```

----End

1.5.8.2 Using Postman to Test IoT Platform APIs

Prerequisites

Before using this method, you need to:

- Obtain the IP address and port number (HTTPS-compliant) provided by the IoT platform for applications.
- Install and run Postman.

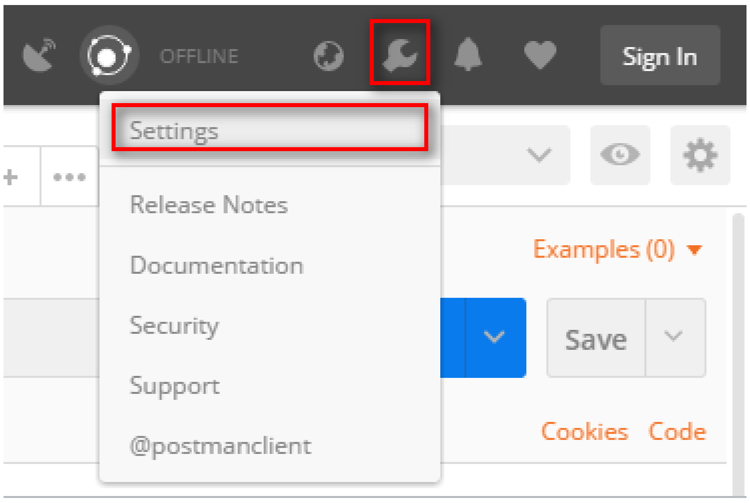
  NOTE

  The Postman installation package is available at **https://www.getpostman.com**.

Configuring Postman

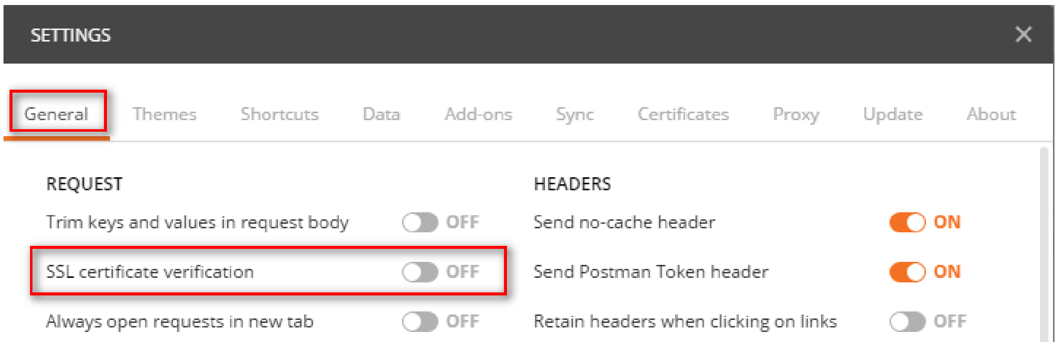
Step 1 Choose **Settings**.

Figure 1-41 Choosing Settings



Step 2 Disable certificate verification so that Postman does not verify the server certificate.

Figure 1-42 Disabling certificate verification



REQUEST

Trim keys and values in request body OFF

SSL certificate verification OFF

Always open requests in new tab OFF

HEADERS

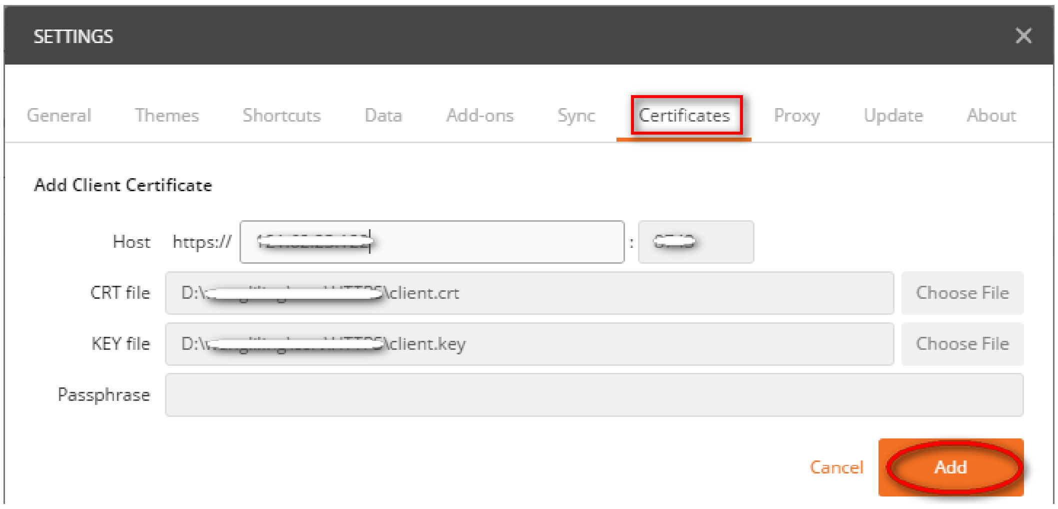
Send no-cache header ON

Send Postman Token header ON

Retain headers when clicking on links OFF

Step 3 Configure the client certificate. Specifically, enter the IP address and port number provided by the IoT platform for applications in the **Host** input box.

Figure 1-43 Configuring the client certificate



SETTINGS

General | Themes | Shortcuts | Data | Add-ons | Sync | Certificates | Proxy | Update | About

Add Client Certificate

Host	https:// [redacted] : [redacted]
CRT file	D:\[redacted]\client.crt
KEY file	D:\[redacted]\client.key
Passphrase	

Cancel | Add

----End

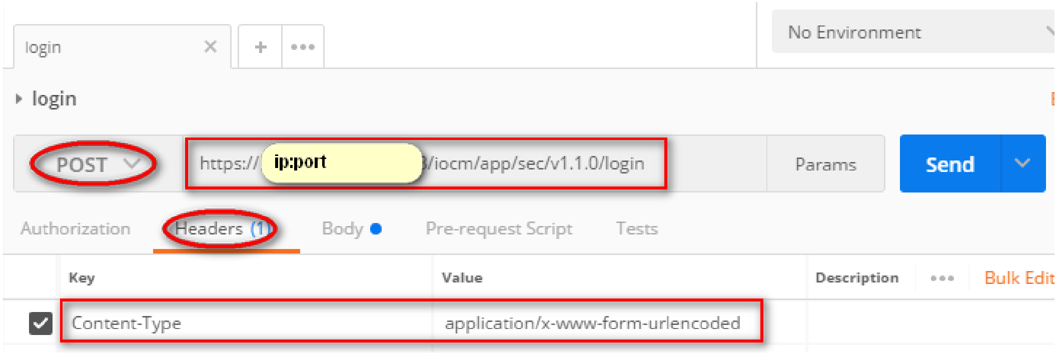
NOTE

client.crt and **client.key** are the client certificate and the private key file.

Debugging the Authentication API

Step 1 Configure the HTTP method, URL, and **Headers** of the authentication API.

Figure 1-44 Configuring the HTTP method, URL, and Headers of the authentication API



login | No Environment

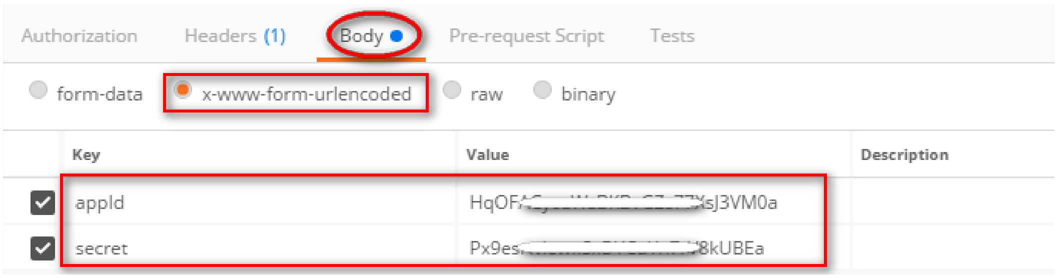
POST https://ip:port/iocm/app/sec/v1.1.0/login | Params | Send

Authorization | Headers (1) | Body | Pre-request Script | Tests

	Key	Value	Description
✓	Content-Type	application/x-www-form-urlencoded	

Step 2 Configure **Body** of the authentication API.

Figure 1-45 Configuring Body of the authentication API



Authorization | Headers (1) | Body | Pre-request Script | Tests

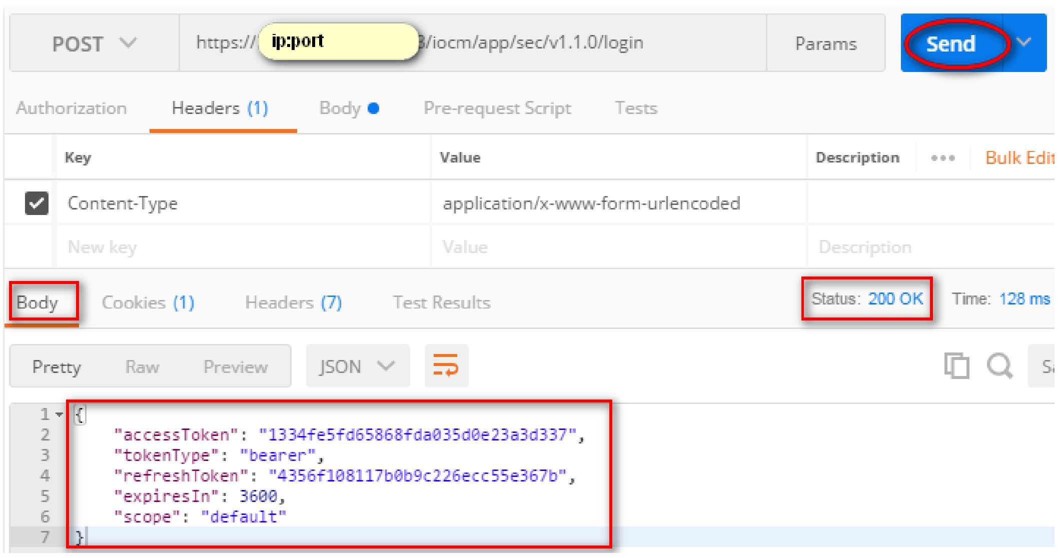
form-data [✓ x-www-form-urlencoded] raw binary

	Key	Value	Description
✓	appId	HqOF[redacted]sJ3VM0a	
✓	secret	Px9es[redacted]8kUBEa	

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

Keep the access token securely. It will be used when other APIs are called.

Figure 1-46 Viewing the response of the Auth API



Key	Value	Description
Content-Type	application/x-www-form-urlencoded	
New key	Value	Description

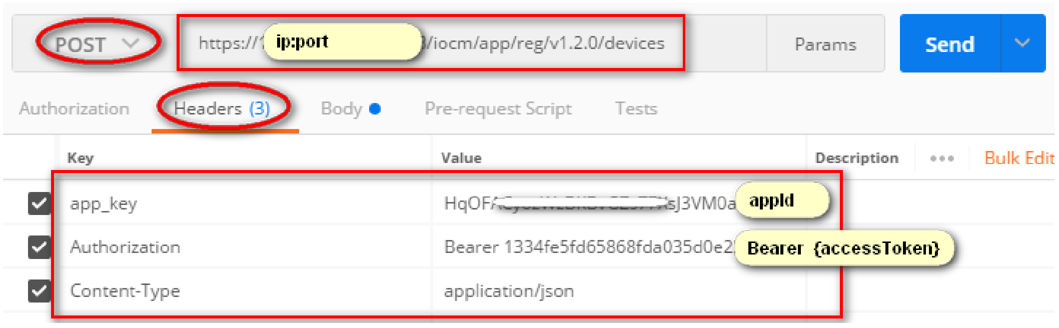
```
{
    "accessToken": "1334fe5fd65868fda035d0e23a3d337",
    "tokenType": "bearer",
    "refreshToken": "4356f108117b0b9c226ecc55e367b",
    "expiresIn": 3600,
    "scope": "default"
}
```

----End

Debugging the API for Registering a Directly Connected Device

Step 1 Configure the HTTP method, URL, and **Headers** of the API for registering a directly connected device.

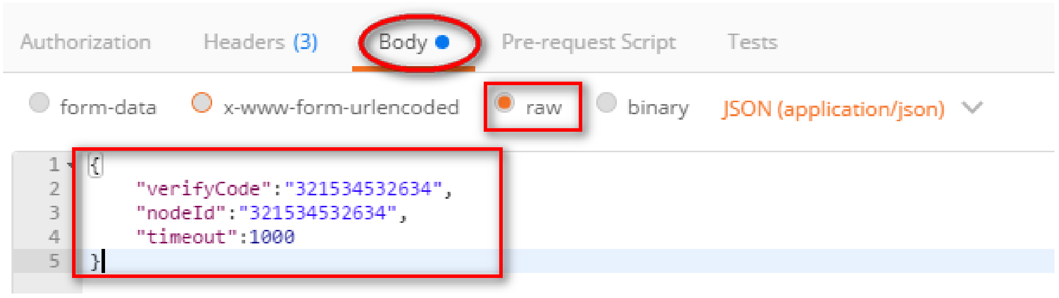
Figure 1-47 Configuring the HTTP method, URL, and Headers of the API for registering a directly connected device



Key	Value	Description
app_key	HqOF[redacted]sJ3VM0a appId	
Authorization	Bearer 1334fe5fd65868fda035d0e2 Bearer {accessToken}	
Content-Type	application/json	

Step 2 Configure **Body** of the API for registering a directly connected device.

Figure 1-48 Configuring Body of the API for registering a directly connected device

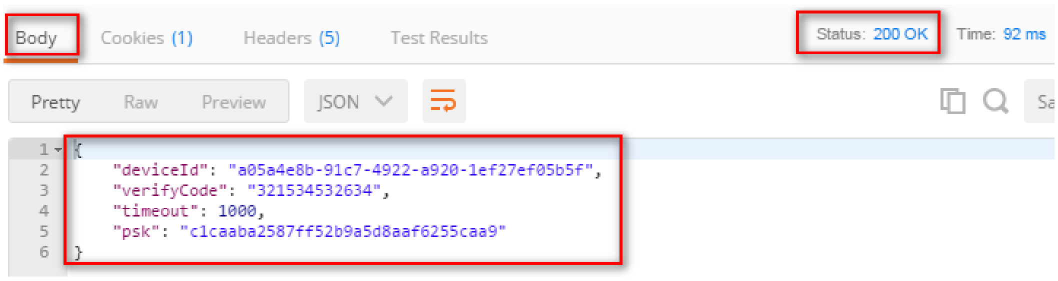


```
{
    "verifyCode":"321534532634",
    "nodeId":"321534532634",
    "timeout":1000
}
```

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

Keep the returned device ID properly. It will be used when other APIs are called.

Figure 1-49 Viewing the response of the API for registering a directly connected device



```
{
    "deviceId": "a05a4e8b-91c7-4922-a920-1ef27ef05b5f",
    "verifyCode": "321534532634",
    "timeout": 1000,
    "psk": "c1caaba2587ff52b9a5d8aaf6255caa9"
}
```

----End

1.5.8.3 CA Certificate

Exporting a CA Certificate

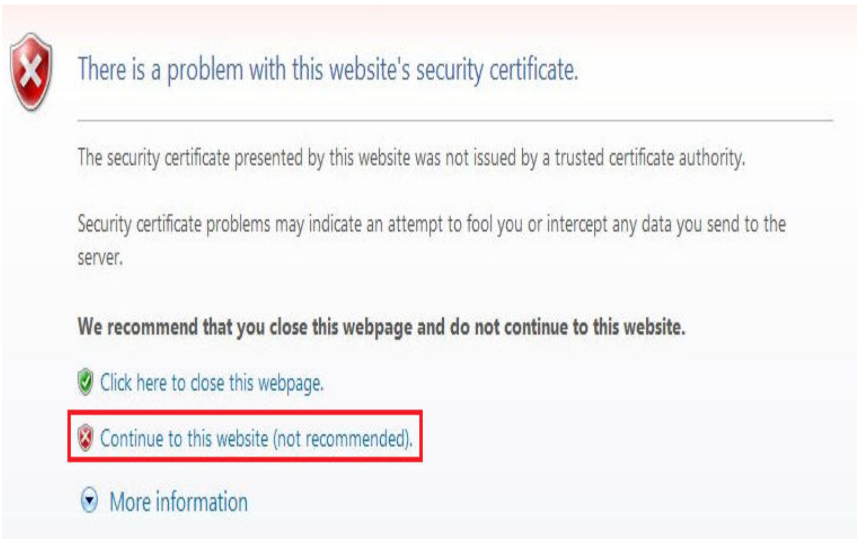
The CA certificate on the application server can be exported as follows:

Step 1 Start a browser, and type the callback address in the address box. Internet Explorer is used as an example.

Step 2 Check the certificate. The methods for checking a self-signed certificate and non-self-signed certificate are different.

- If the callback address uses a self-signed certificate, the message "There is a problem with this website's security certificate" is displayed. Choose **Continue to this website (not recommended).** > **Certificate Error** > **View certificates**.

Figure 1-50 Self-signed certificate callback address prompt



There is a problem with this website's security certificate.

The security certificate presented by this website was not issued by a trusted certificate authority.

Security certificate problems may indicate an attempt to fool you or intercept any data you send to the server.

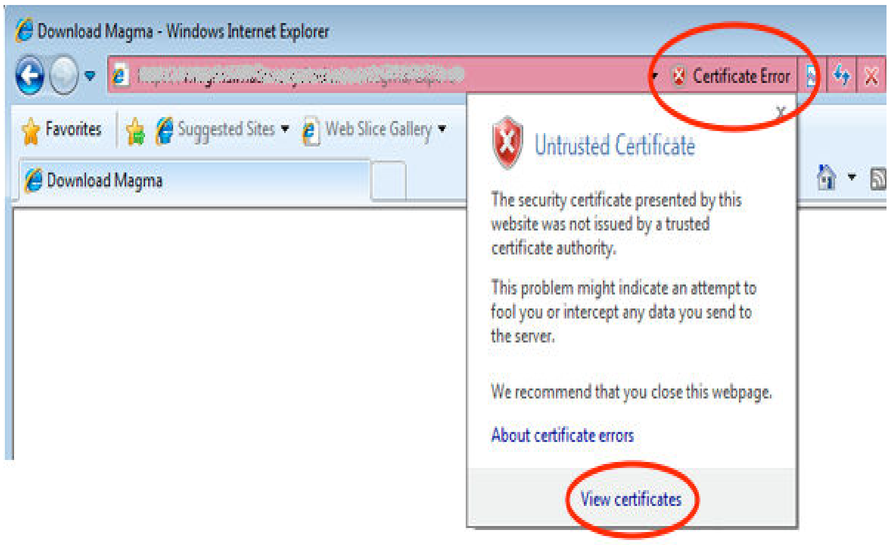
We recommend that you close this webpage and do not continue to this website.

Click here to close this webpage.

Continue to this website (not recommended).

More information

Figure 1-51 Checking the self-signed certificate



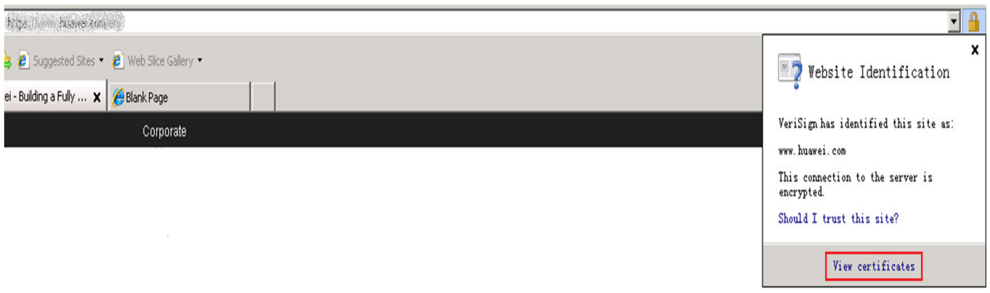
The security certificate presented by this website was not issued by a trusted certificate authority.

This problem might indicate an attempt to fool you or intercept any data you send to the server.

We recommend that you close this webpage.

- If the callback address is not a self-signed certificate, choose **Security Report** > **View certificates**.

Figure 1-52 Checking the non-self-signed certificate



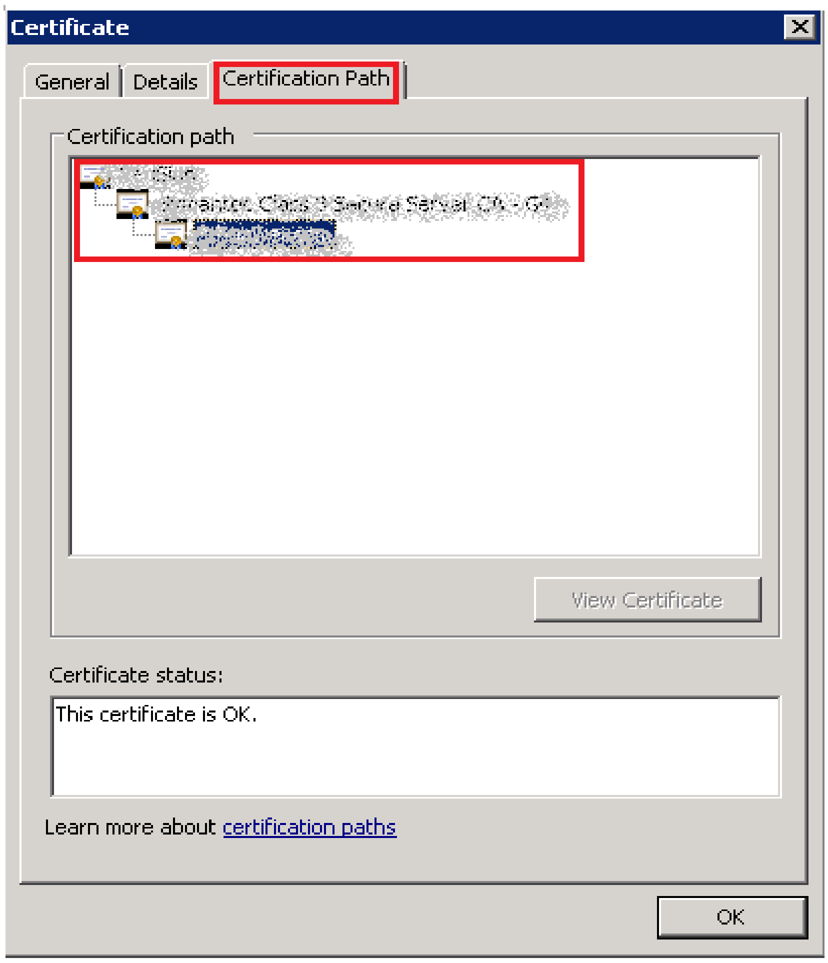
VeriSign has identified this site as:

www.huawei.com

This connection to the server is encrypted.

Step 3 Check the certificate level on the **Certification Path** tab page. The current certificate level is the last level of the certificate.

Figure 1-53 Certificate Directory



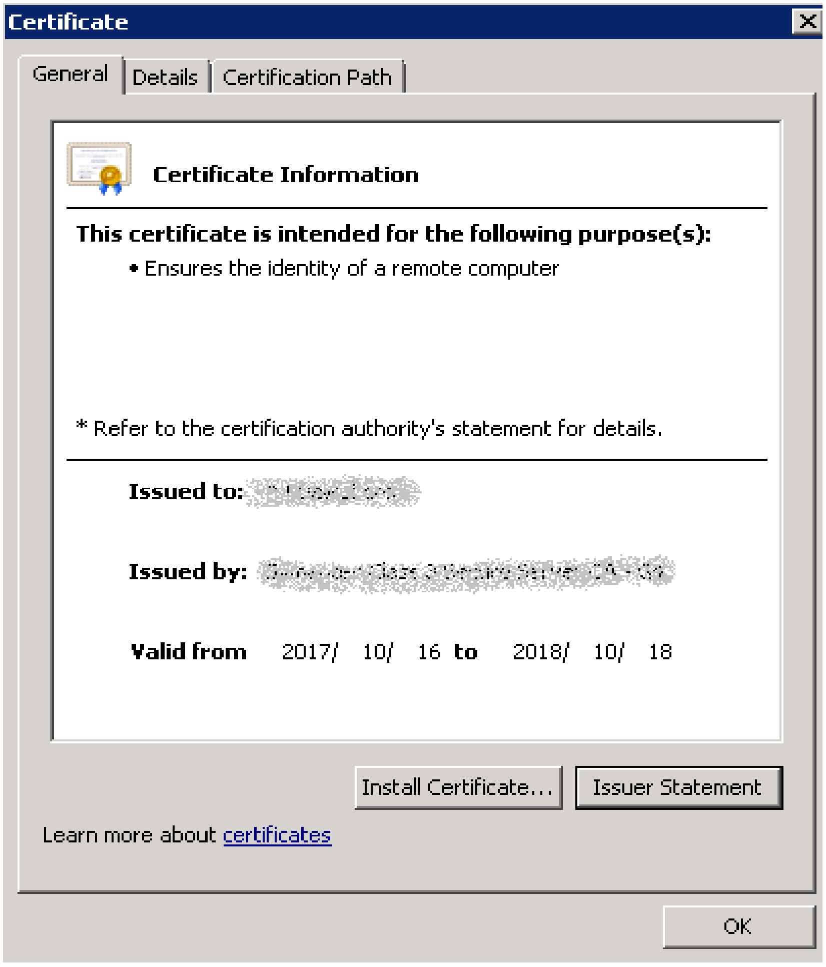
[redacted]

Certificate status:

This certificate is OK.

Learn more about certification paths

Figure 1-54 Common certificate information


Certificate

General | Details | Certification Path

Certificate Information

This certificate is intended for the following purpose(s):

• Ensures the identity of a remote computer

* Refer to the certification authority's statement for details.

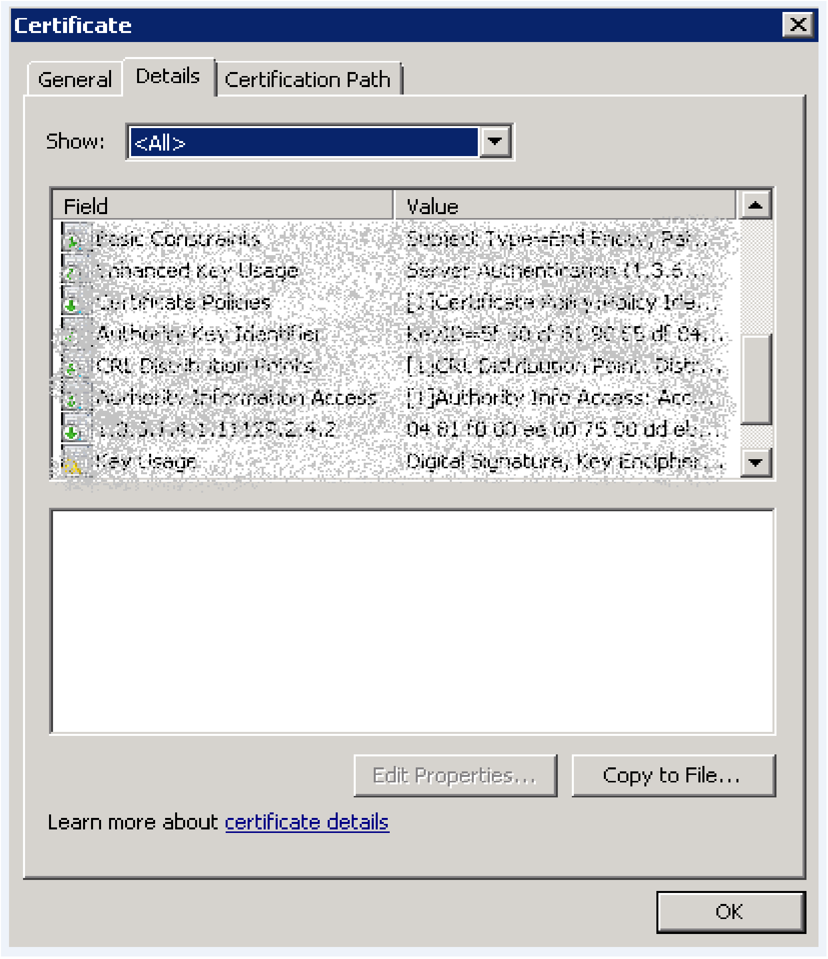
Issued to:

Issued by:

Valid from 2017/ 10/ 16 to 2018/ 10/ 18

Step 4 On the **Details** tab page, click **Copy to File**, and select **Base-64 encoded X.509 (.CER)** to export the certificate of the current level based on the certificate export wizard.

Figure 1-55 Detailed certificate information

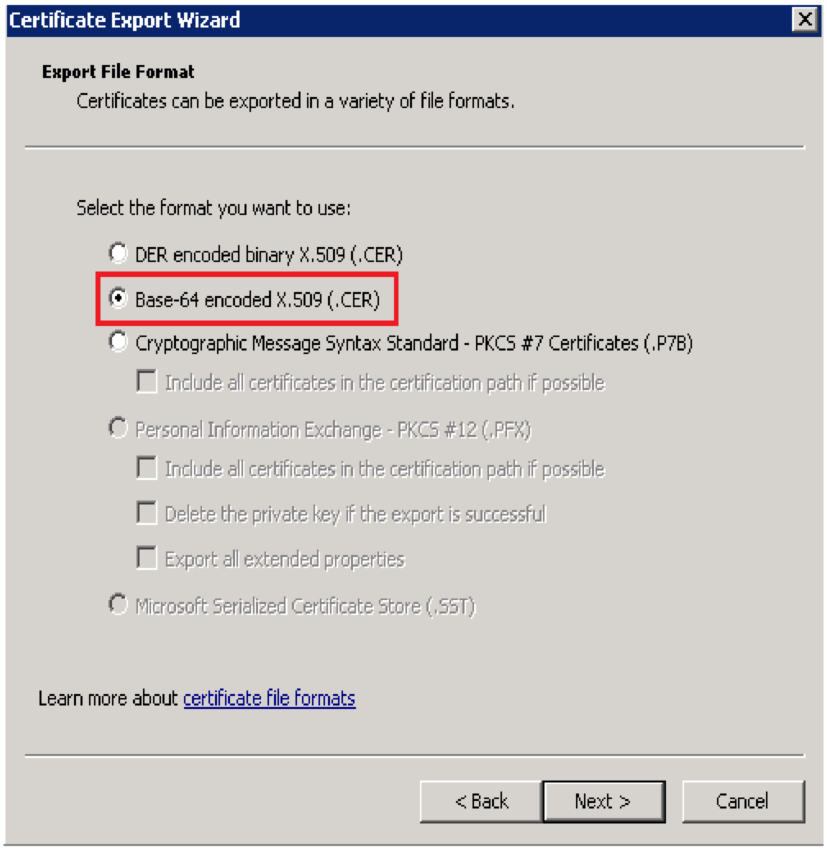


Show: <All>

Field	Value
Basic Constraints	Subject Type=End Entity, Pat...
Enhanced Key Usage	Server Authentication (1.3.6...
Certificate Policies	[1]Certificate Policy:Policy Ide...
Authority Key Identifier	KeyID=5f 30 cf 91 90 55 df 84...
CRL Distribution Points	[1]CRL Distribution Point: Distr...
Authority Information Access	[1]Authority Info Access: Acc...
1.3.6.1.4.1.11129.2.4.2	04 81 f0 00 ee 00 75 00 dd eb...
Key Usage	Digital Signature, Key Encipher...

Learn more about certificate details

Figure 1-56 Selecting the certificate export format



Certificate Export Wizard

Export File Format

Certificates can be exported in a variety of file formats.

Select the format you want to use:

- DER encoded binary X.509 (.CER)
- [✓ Base-64 encoded X.509 (.CER)]
- Cryptographic Message Syntax Standard - PKCS #7 Certificates (.P7B)
  - Include all certificates in the certification path if possible
- Personal Information Exchange - PKCS #12 (.PFX)
  - Include all certificates in the certification path if possible
  - Delete the private key if the export is successful
  - Export all extended properties
- Microsoft Serialized Certificate Store (.SST)

Learn more about certificate file formats

Step 5 Double-click the upper-level certificate. In the displayed dialog box, choose **Details** > **Copy to File** > **Base-64 encoded X.509 (.CER)** to export the upper-layer certificate by following instructions provided by the certificate export wizard.

Step 6 Repeat **Step 5** until all levels of certificates are exported.

Step 7 Use the text editor to combine all the exported certificates into a file.

NOTE

No newline character exists between the combined files.

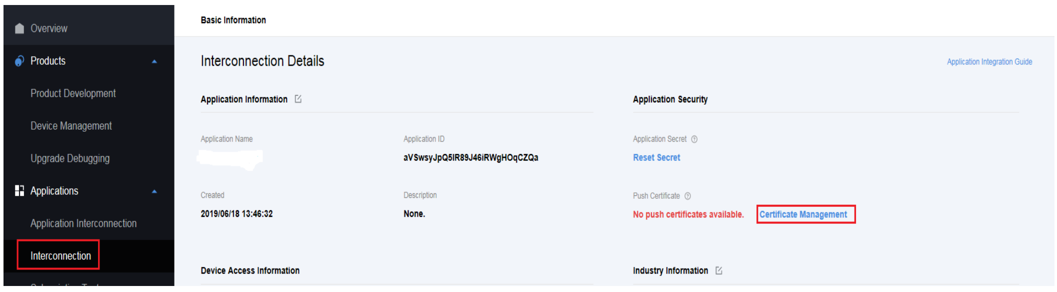
Figure 1-57 Combining certificates

```
pLkJkLQYWBSHuxOizGdwCjw1mAT5G9+443fNDsgN3BAAAAFc8uXxRAAABAMARjBE
AiA+PkSvZOvIBmxaBLXLceHH1NqW+Mcrz+xtXNmJO12E1QIgfVeG4xeuWrb7bpoU
smR+LStIG1TPCwimn3JRE3we/fYwDQYJKoZIhvcNAQELBQADggEBADjrCz8a7cax
h7vpyuUFZ/fiKBHE7VLqfppgf3XYNBoqh21qM6gTGzdiSeZj+vx+KOUn38f080Rg
N2aEkag3n03cufIXR8Yn8haXcusz5PONS1MQnN5rZBwpZ8obIti08KGOh51gHQ+s
S1oX/j8nDDCQgrNkcG2A78nUT+VxGGENxnPmqajP/O2h/kg02qjcnPoj6E1mm/At
5dWWANX374yS7c0fgLZZ1mfZoIqooaRxsSJ15RzyRNU3Bzv5CZCJCGYFqC3RS28Q
vTCjde7TMsAQiWkZ97IK1UMXdbHManm7K85aWcG4Wg8isr9d2GPUZYgcUSc8KfWY
aP5MzoeU6ug=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFODCCBCCgAwIBAgIQUT+5dDhwtzRAQY0wkwaZ/zANBgkqhkiG9w0BAQsFADCB
yjELMAkGA1UEBhMCVVMxFzAVBgNVBAoTD1Z1cm1TaWduLCBJbmMuMR8wHQYDVQQL
ExZWZXJpU21nbiBUcnVzdCBOZXR3b3JrMTowOAYDVQQLEzEoYykgMjAwNiBWZXJp
U21nbiwgSW5jLiAtIEZvciBhdXRob3JpemVkIHVzZSBvbmx5MUUwQwYDVQQDEzxW
ZXJpU21nbiBDbGFzcyAzIFB1YmxpYyBQcm1tYXJ5IEN1cnRpZm1jYXRpb24gQXV0
```

Step 8 Change the file name suffix to **pem**.

Step 9 On the Developer Center, choose **Interconnection** > **Application Security** > **Push Certificate** to upload the certificate to the IoT platform.

Figure 1-58 Uploading a Certificate



Application Name [redacted]
Application ID aVSwsyJpQ5IR89J46iRWgHOqCZQa
Application Secret Reset Secret
Created 2019/06/18 13:46:32
Description None.
Push Certificate No push certificates available. Certificate Management

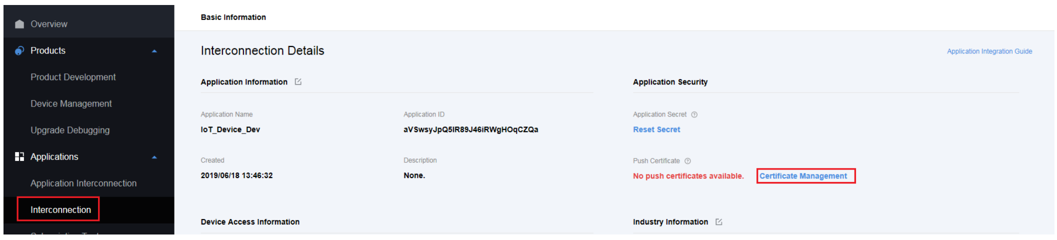
----End

Uploading a CA Certificate

The CA certificate of the application server must be uploaded to the IoT platform for the IoT platform to push HTTPS messages to the NA server. The CA certificate can be uploaded through the Developer Center or the SP portal.

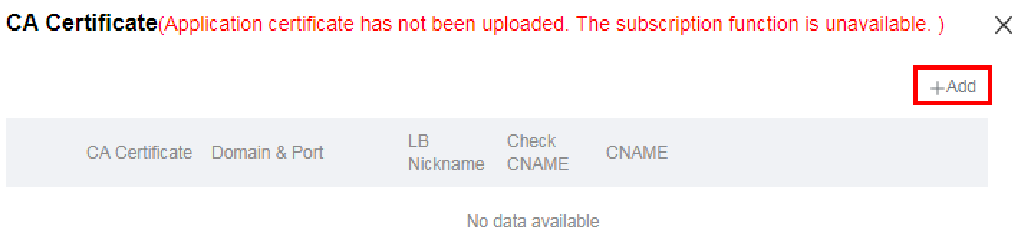
Uploading the CA Certificate Through the Developer Center

Step 1 Choose **Applications** > **Interconnection**. In the **Push Certificate** area, click **Certificate Management**.



Application Name IoT_Device_Dev
Application ID aVSwsyJpQ5IR89J46iRWgHOqCZQa
Application Secret Reset Secret
Created 2019/06/18 13:46:32
Description None.
Push Certificate No push certificates available. Certificate Management

Step 2 The **CA Certificate** dialog box is displayed. Check whether the CA certificate has been uploaded. If not, click **Add**.



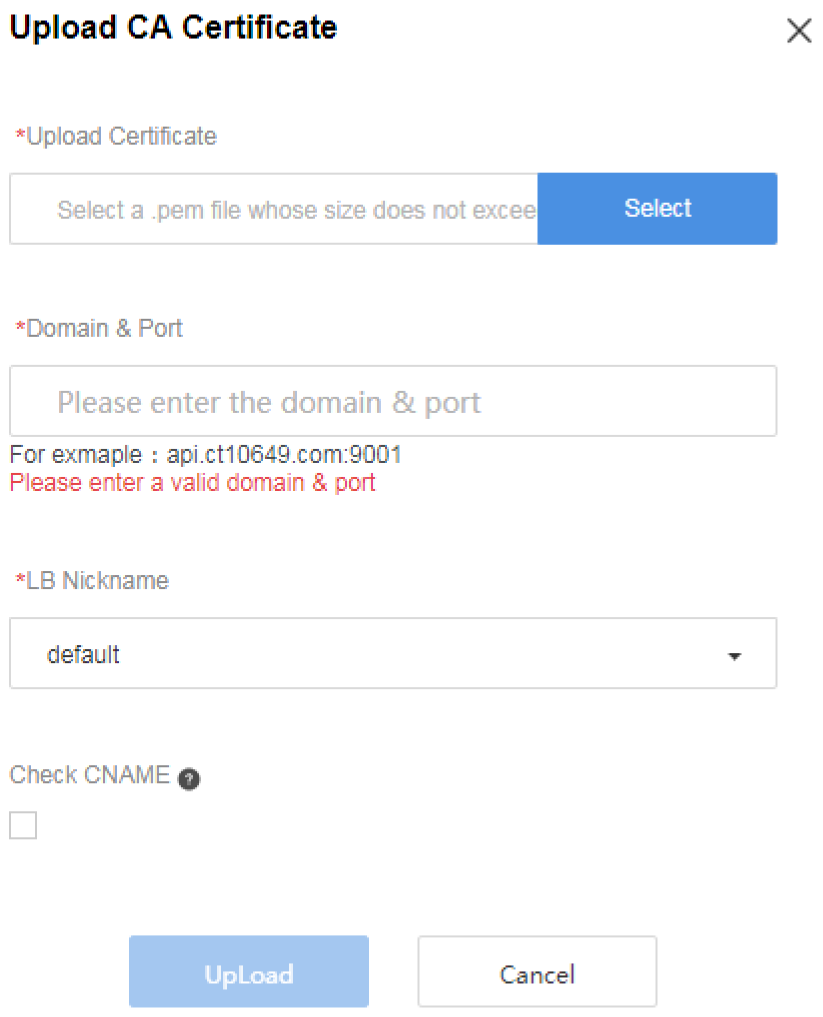
CA Certificate(Application certificate has not been uploaded. The subscription function is unavailable.)

+Add

CA Certificate	Domain & Port	LB Nickname	Check CNAME	CNAME

No data available

Step 3 In the displayed **Upload CA Certificate** dialog box, select the certificate file, set parameters, and click **Upload**.



Upload CA Certificate

*Upload Certificate

Select a .pem file whose size does not excee

Select

*Domain & Port

Please enter the domain & port

For exmaple : api.ct10649.com:9001

Please enter a valid domain & port

*LB Nickname

default

Check CNAME

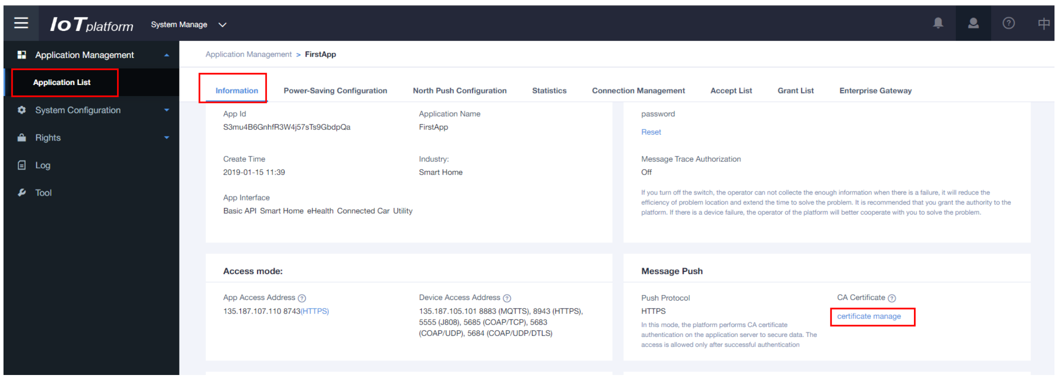
UpLoad

Cancel

----End

Uploading the CA Certificate on the Management Portal

Step 1 Choose **System Manage** from the upper navigation bar, select **Application Management** > **Application List**, select an application, and click **certificate manage** on the **Information** tab page.



App Id
S3mu4B6GnhfR3W4j57sTs9GbdpQa

Application Name
FirstApp

Create Time
2019-01-15 11:39

Industry:
Smart Home

App Interface
Basic API Smart Home eHealth Connected Car Utility

password

Message Trace Authorization
Off

If you turn off the switch, the operator can not collecte the enough information when there is a failure, it will reduce the efficiency of problem location and extend the time to solve the problem. It is recommended that you grant the authority to the platform. If there is a device failure, the operator of the platform will better cooperate with you to solve the problem.

Access mode:

App Access Address
135.187.107.110 8743(HTTPS)

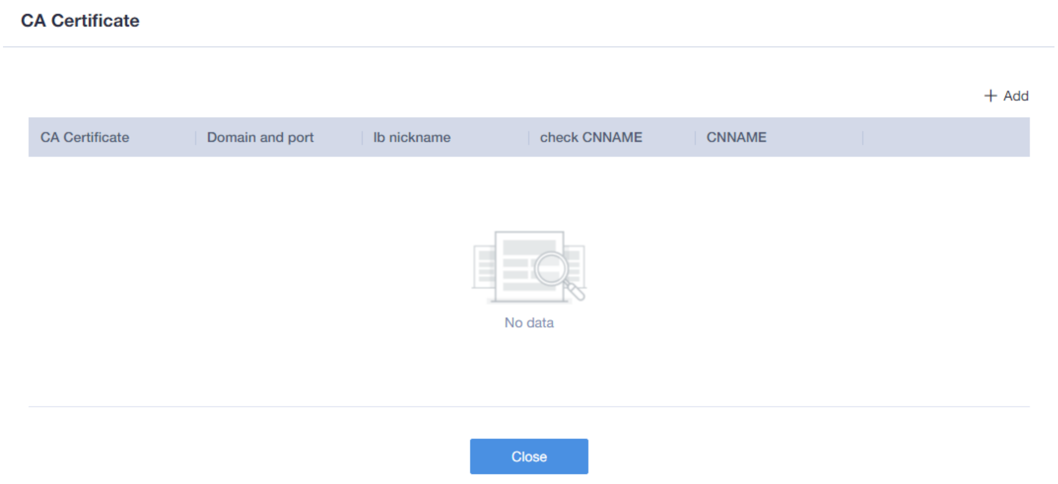
Device Access Address
135.187.105.101 8883 (MQTTS), 8943 (HTTPS), 5555 (J808), 5685 (COAP/TCP), 5683 (COAP/UDP), 5684 (COAP/UDP/DTLS)

Message Push

Push Protocol
HTTPS
In this mode, the platform performs CA certificate authentication on the application server to secure data. The access is allowed only after successful authentication

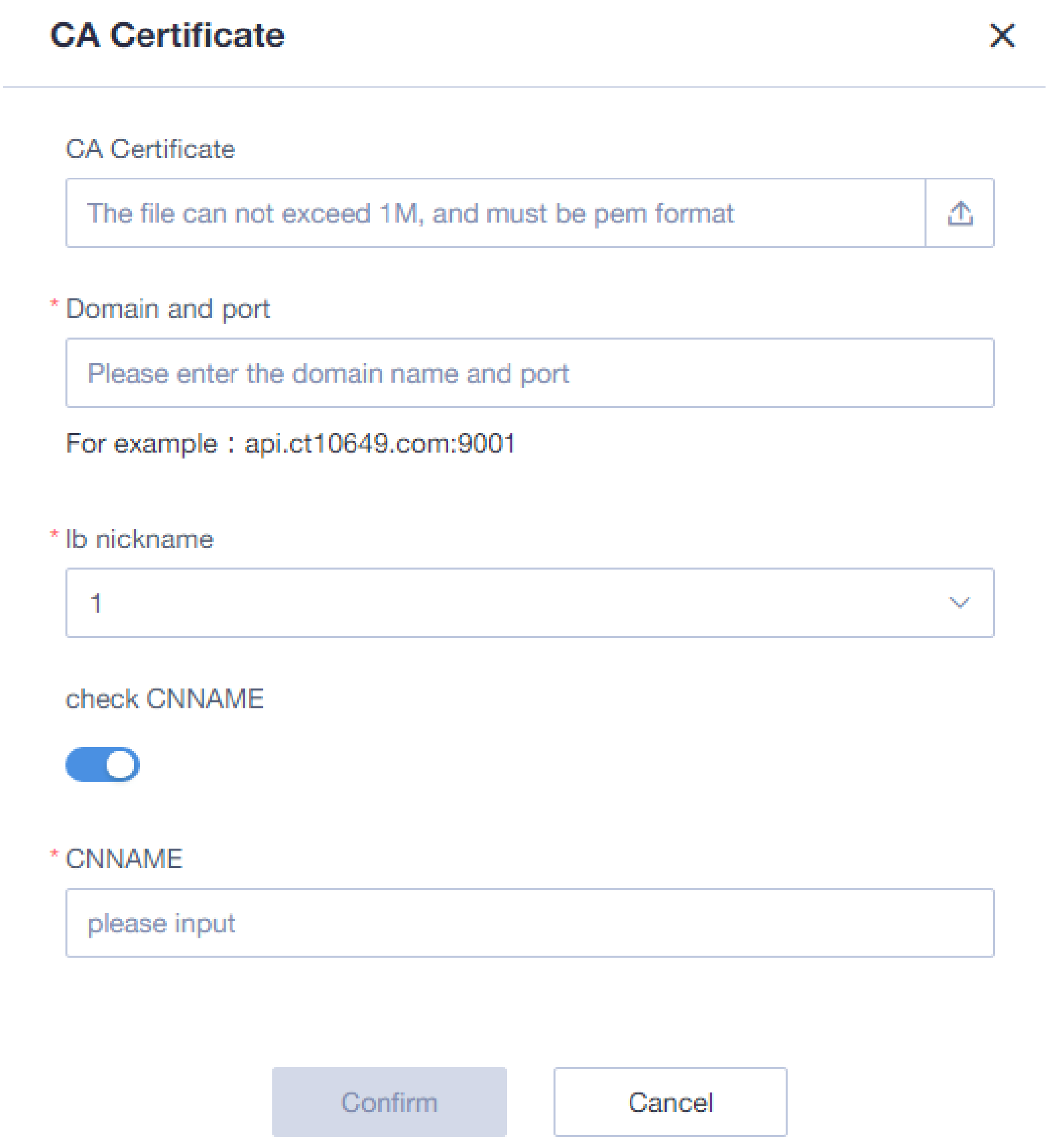
CA Certificate

Step 2 The **CA Certificate** dialog box is displayed. Check whether the CA certificate has been uploaded. If not, click **Add**.



CA Certificate	Domain and port	lb nickname	check CNNAME	CNNAME

Step 3 In the displayed **CA Certificate** dialog box, select the certificate file, set parameters, and click **Confirm**.


CA Certificate

CA Certificate

The file can not exceed 1M, and must be pem format

* Domain and port

Please enter the domain name and port

For example：api.ct10649.com:9001

* lb nickname

1

check CNNAME

* CNNAME

please input

----End

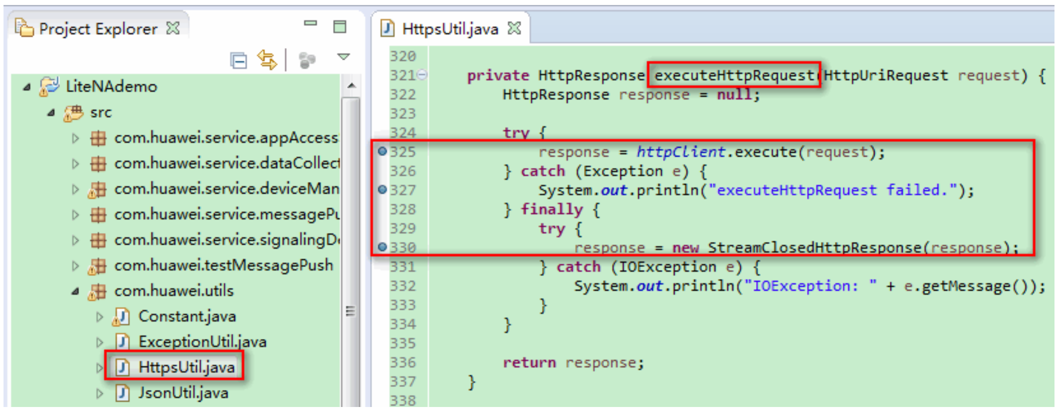
1.5.8.4 Performing Single-Step Debugging

To view requests sent by applications and responses from the IoT platform in a more intuitive manner, use the breakpoint debugging method of Eclipse. If you use the Postman test interface, see **Using Postman to Test IoT Platform APIs**.

Step 1 Set a breakpoint at the code where HTTP or HTTPS messages are sent.

For example, set three breakpoints for the **executeHttpRequest** method in the sample code **HttpsUtil.java**. (Set the breakpoints according to the actual situation of your code.)

Figure 1-59 Setting a breakpoint


Project Explorer
LiteNAdemo
src
com.huawei.service.appAccess
com.huawei.service.dataCollect
com.huawei.service.deviceMan
com.huawei.service.messagePu
com.huawei.service.signalingD
com.huawei.testMessagePush
com.huawei.utils
Constant.java
ExceptionUtil.java
HttpsUtil.java
JsonUtil.java
HttpsUtil.java

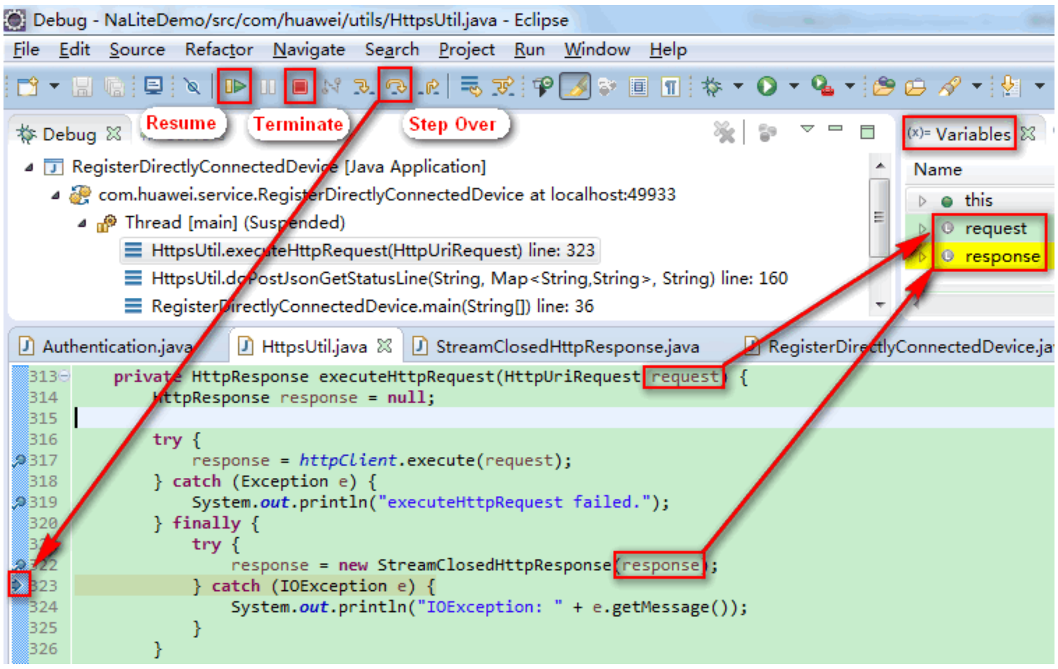
```
    private HttpResponse executeHttpRequest(HttpUriRequest request) {
        HttpResponse response = null;

        try {
            response = httpClient.execute(request);
        } catch (Exception e) {
            System.out.println("executeHttpRequest failed.");
        } finally {
            try {
                response = new StreamClosedHttpResponse(response);
            } catch (IOException e) {
                System.out.println("IOException: " + e.getMessage());
            }
        }

        return response;
    }
```

Step 2 Right-click the class to be debugged based on the project type, for example, **Authentication.java**, and choose **Debug As** > **Java Application**.

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 corresponding variable in the **Variables** window, such as the sent messages and the response messages of the IoT platform.

Figure 1-60 Performing single-step debugging


Debug - NaLiteDemo/src/com/huawei/utils/HttpsUtil.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Resume
Terminate
Step Over
Variables
Name
this
request
response
Authentication.java
HttpsUtil.java
StreamClosedHttpResponse.java
RegisterDirectlyConnectedDevice.ja

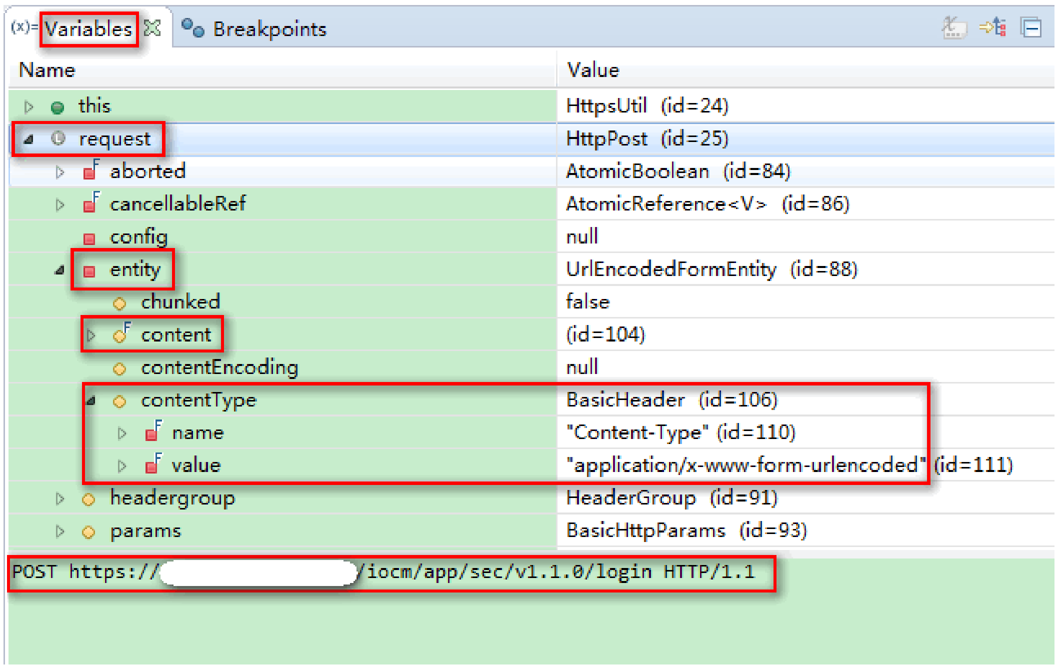
```
RegisterDirectlyConnectedDevice [Java Application]
  com.huawei.service.RegisterDirectlyConnectedDevice at localhost:49933
    Thread [main] (Suspended)
      HttpsUtil.executeHttpRequest(HttpUriRequest) line: 323
      HttpsUtil.doPostJsonGetStatusLine(String, Map<String,String>, String) line: 160
      RegisterDirectlyConnectedDevice.main(String[]) line: 36
```

```
    private HttpResponse executeHttpRequest(HttpUriRequest request) {
        HttpResponse response = null;

        try {
            response = httpClient.execute(request);
        } catch (Exception e) {
            System.out.println("executeHttpRequest failed.");
        } finally {
            try {
                response = new StreamClosedHttpResponse(response);
            } catch (IOException e) {
                System.out.println("IOException: " + e.getMessage());
            }
        }
```

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

When the **request** variable is selected, the URLs of the requests sent by the applications are displayed in the content area in the lower part of the pane, and the content the requests is displayed in the **entity** area.

Figure 1-61 Expanding the request variable

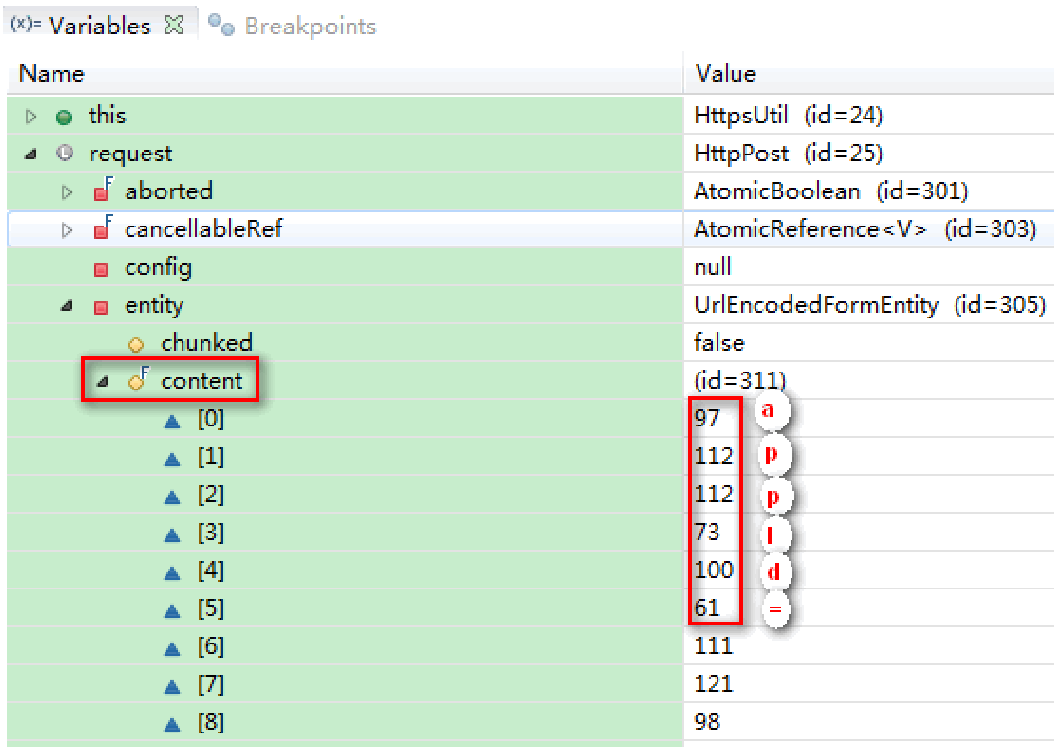


Name	Value
this	HttpsUtil (id=24)
request	HttpPost (id=25)
aborted	AtomicBoolean (id=84)
cancellableRef	AtomicReference<V> (id=86)
config	null
entity	UrlEncodedFormEntity (id=88)
chunked	false
content	(id=104)
contentEncoding	null
contentType	BasicHeader (id=106)
name	"Content-Type" (id=110)
value	"application/x-www-form-urlencoded" (id=111)
headergroup	HeaderGroup (id=91)
params	BasicHttpParams (id=93)

```
POST https://[redacted]/iocm/app/sec/v1.1.0/login HTTP/1.1
```

The application ID (appId) and application key (secret) are contained in the **content** field and are represented by decimal ASCII codes. You need to convert them into letters and symbols according to the ASCII code table.

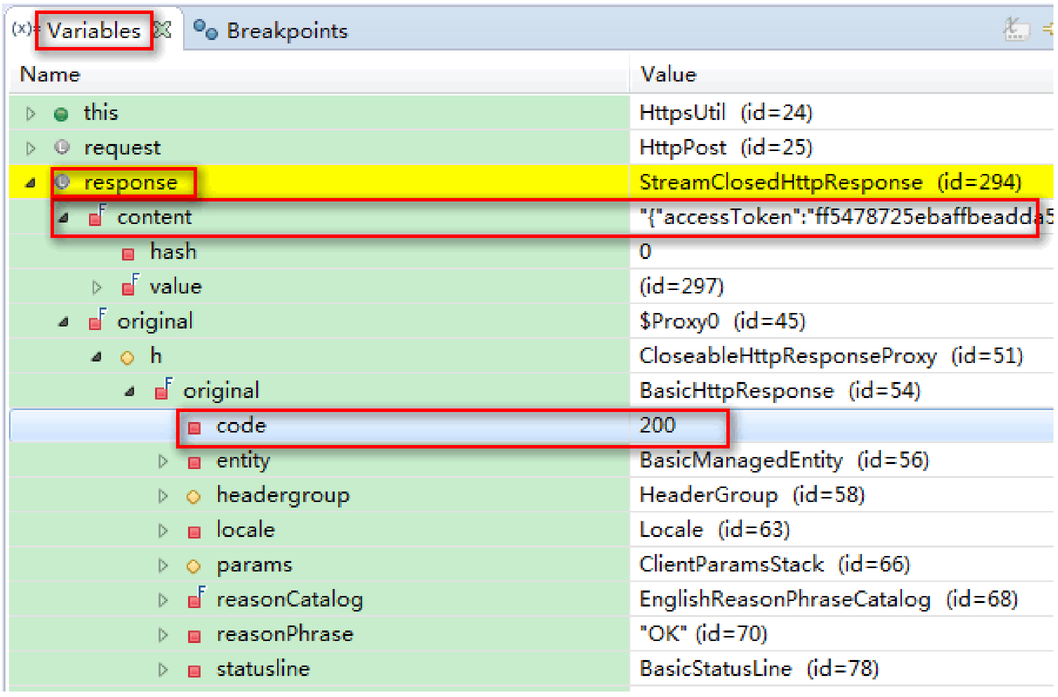
Figure 1-62 Viewing the content field



Name	Value
this	HttpsUtil (id=24)
request	HttpPost (id=25)
aborted	AtomicBoolean (id=301)
cancellableRef	AtomicReference<V> (id=303)
config	null
entity	UrlEncodedFormEntity (id=305)
chunked	false
content	(id=311)
[0]	97
[1]	112
[2]	112
[3]	73
[4]	100
[5]	61
[6]	111
[7]	121
[8]	98

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

Figure 1-63 Expanding the response variable



Variables | Breakpoints

Name	Value
this	HttpsUtil (id=24)
request	HttpPost (id=25)
response	StreamClosedHttpResponse (id=294)
content	"{"accessToken":"ff5478725ebaffbeadda5
hash	0
value	(id=297)
original	$Proxy0 (id=45)
h	CloseableHttpResponseProxy (id=51)
original	BasicHttpResponse (id=54)
code	200
entity	BasicManagedEntity (id=56)
headergroup	HeaderGroup (id=58)
locale	Locale (id=63)
params	ClientParamsStack (id=66)
reasonCatalog	EnglishReasonPhraseCatalog (id=68)
reasonPhrase	"OK" (id=70)
statusline	BasicStatusLine (id=78)

NOTE

In the code example, all classes other than **Authentication.java** call the Auth API in the first step. Therefore, when performing single-step debugging on a class other than **Authentication.java**, view the variable content when the program runs for the second time to the position where the breakpoint is set.

----End

1.6 Developing a Device

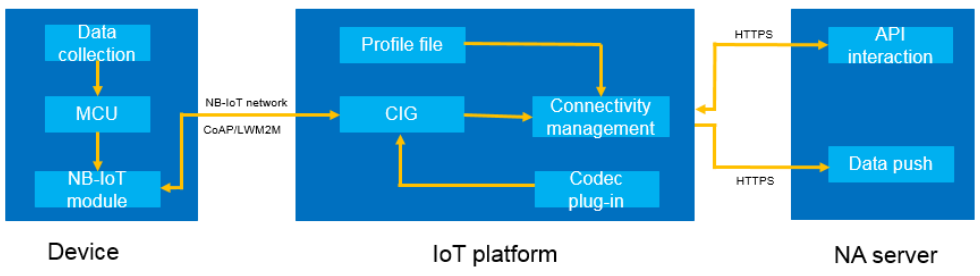
1.6.1 LWM2M/CoAP Device Integration

1.6.1.1 Device Integration

In CoAP or LWM2M access scenarios, devices can be connected to the IoT platform by integrating NB-IoT modules or LiteOS SDK.

Integrating NB-IoT Modules

Devices integrated with NB-IoT modules can connect to the IoT platform through NB-IoT networks.



Features	● Wide coverage: The gain is 20 dB higher than that of LTE. ● Low power consumption: The solution focuses on applications with small data volume at a low rate. ● Massive amounts of connections: A single sector supports a maximum of 50,000 connections. ● Low cost: NB-IoT chipsets or modules are cost-effective for its low rate, low power consumption, and low bandwidth.
Scenarios	Low requirements on data timeliness, small data packets, fixed locations, and power supply from batteries. For example, smart metering and smart street lamp.
Applicable Networks	● NB-IoT network: constructed by carriers ● NB-IoT SIM card: purchased from NB-IoT network carriers ● NB-IoT module: purchased from the module manufacturers
Communication Protocols	CoAP/LWM2M
Related Resources	Obtain more information and support from the module manufacturer.

Integrating LiteOS SDK

LiteOS SDK is a lightweight SDK integrated on the device. Its features are as follows:

Features	● Protocols and security details are shielded. Users can focus on their applications without paying attention to the implementation of protocols and security policies. ● An adaptation layer is provided. Users can migrate LiteOS SDK by adapting only a few interfaces. ● Data reported by devices can be cached and retransmission and acknowledgment mechanisms are provided to ensure data reporting reliability. ● Firmware upgrade, resumable download, and integrity protection for firmware packages are supported. ● Security and non-security connection modes are supported.

Running Environment Requirements	RAM > 32KB FLASH > 128KB
Applicable Networks	NB-IoT, 2G/3G/4G, and wired network
Communication Protocols	CoAP and LWM2M
Related Resources	For details about how to integrate the LiteOS SDK, see **LiteOS SDK Integration Development Guide**.

AT Command

AT commands are used to control devices. The following AT commands are for reference only. Obtain the command set from the corresponding module manufacturer.

AT Command	Function	Remarks
AT+CMEE=1	To query an error.	Standard AT command.
AT+CFUN=0	To power off a device. Shut down a device before setting the IMEI and IP address of the IoT platform.	Standard AT command.
AT+CGSN=1	To query an IMEI. The IMEI is a type of device ID. When an NA server calls an API to register a device, **nodeId** and **verifyCode** must be set to the IMEI.	Standard AT command.
AT +NTSETID=1,xxxx	xxxx indicates an IMEI. If an IMEI is not found, you can set an IMEI that is unique. The IMEI is a type of device ID. When an NA server calls an API to register a device, **nodeId** and **verifyCode** must be set to the IMEI if the device uses a HiSilicon chipset. If the device uses a Qualcomm chipset, **nodeId** and **verifyCode** must be set to **urn:imei:IMEI**.	Proprietary AT command of the HiSilicon chipset, which stores the IMEI in the flash memory. This parameter is used when the NA server registers with the IoT platform. Other chipset or module manufacturers can refer to this parameter.

AT Command	Function	Remarks
AT +NCDP="IP","port"	Set the IP address and port number of the IoT platform connected to the device. 5683 is a non-encrypted port and 5684 is a DTLS encrypted port.	Proprietary AT command of the HiSilicon chipset, which stores the IP address and port number in the flash memory. This parameter is used when the NA server registers with the IoT platform. Other chipset or module manufacturers can refer to this parameter.
AT+CFUN=1	To power on a device.	Standard AT command.
AT+NBAND= frequency band	To set the frequency band.	Proprietary AT command of the HiSilicon chipset, which stores the frequency band in the flash memory. This parameter is used when the device is connected to a network. Other chipset or module manufacturers can refer to this parameter.
AT +CGDCONT=1,"IP" ,"CTNB"	To set the APN of the core network. The APN is related to the sleep and keep-alive modes of a device and must be confirmed with the carrier.	Standard AT command.
AT+CGATT=1	To access a network.	Standard AT command.
AT+CGPADDR	To obtain the IP address of a device.	Standard AT command.

AT Command	Function	Remarks
AT+NMGS=x,xxxx	To send downstream data. The first parameter indicates the number of bytes, and the second parameter indicates the reported hexadecimal stream.	Proprietary AT command of the HiSilicon chipset. Data transmitted for the first time is used for device registration, and after a device is registered, only data is sent. Other chipset or module manufacturers can refer to this parameter.
AT+NQMGR	To receive downstream data.	Proprietary AT command of the HiSilicon chipset. It is used to query the amount of data that can be received in the receive buffer, the total number of received messages, and the number of discarded messages. Other chipset or module manufacturers can refer to this parameter.
AT+NMGR	To read data.	Proprietary AT command of the HiSilicon chipset. The command is used to read data received from the IoT platform (LWM2M server). Other chipset or module manufacturers can refer to this parameter.

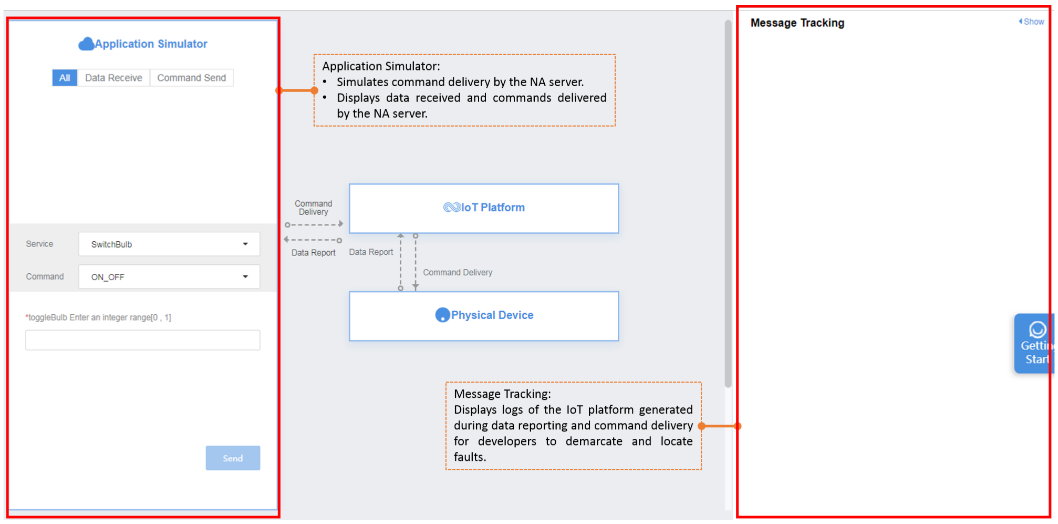
1.6.1.2 Device Testing

Overview

Online test supports device simulation and application simulation. It offers scenarios such as data reporting and command delivery to test devices, profiles, and plug-ins.

Developers can use physical or virtual devices for online test.

- When the device development is complete but the application development is not, developers can add physical devices and use the application simulator to test devices, profiles, and plug-ins. The structure of physical device testing interface is as follows:



Application Simulator:
- Simulates command delivery by the NA server.
- Displays data received and commands delivered by the NA server.

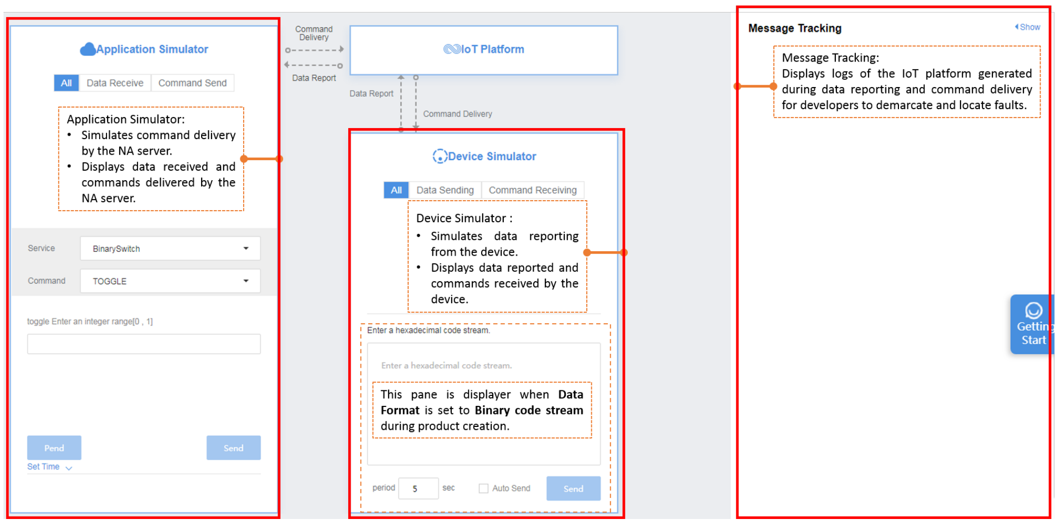
Service: SwitchBulb

Command: ON_OFF

*toggleBulb Enter an integer range[0 , 1]

Message Tracking:
Displays logs of the IoT platform generated during data reporting and command delivery for developers to demarcate and locate faults.

- When both device development and application development are not completed, developers can create virtual devices and use the application simulator and device simulator to test profiles and plug-ins. The structure of virtual device testing interface is as follows:



Application Simulator:
- Simulates command delivery by the NA server.
- Displays data received and commands delivered by the NA server.

Service: BinarySwitch

Command: TOGGLE

toggle Enter an integer range[0 , 1]

Device Simulator :
- Simulates data reporting from the device.
- Displays data reported and commands received by the device.

Enter a hexadecimal code stream.

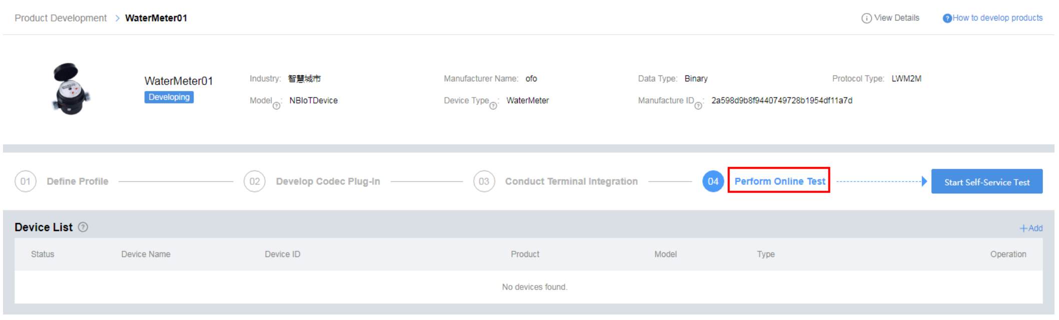
This pane is displayer when **Data Format** is set to **Binary code stream** during product creation.

period 5 sec Auto Send Send

Message Tracking:
Displays logs of the IoT platform generated during data reporting and command delivery for developers to demarcate and locate faults.

Using a Physical Device for Online Test

Step 1 In the product development space, click **Perform Online Test**.


Product Development > WaterMeter01

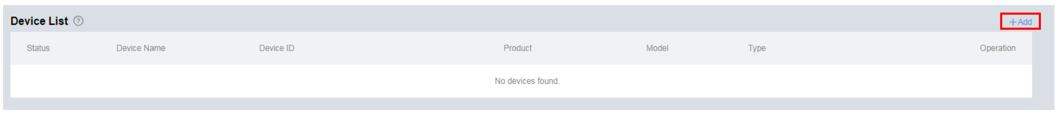
WaterMeter01 Developing

Industry: 智慧城市
Model: NBIoTDevice
Manufacturer Name: ofo
Device Type: WaterMeter
Data Type: Binary
Manufacture ID: 2a598d9b8f9440749728b1954df11a7d
Protocol Type: LWM2M

Device List

Status	Device Name	Device ID	Product	Model	Type	Operation
No devices found.						

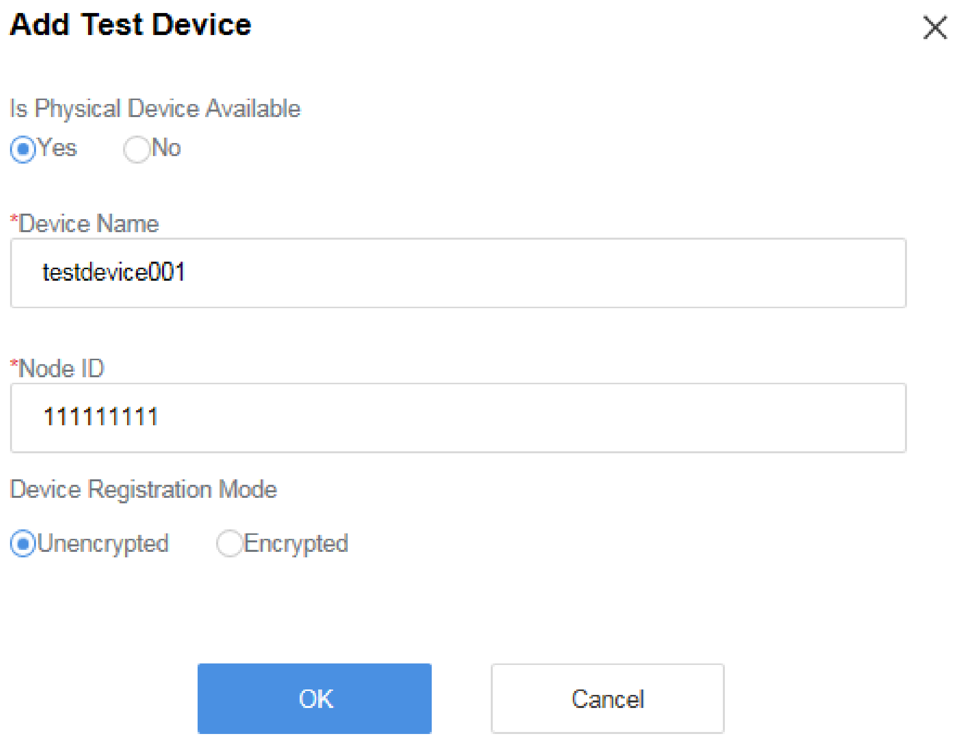
Step 2 Click **Add** at the row where **Device List** resides.


Device List

Status	Device Name	Device ID	Product	Model	Type	Operation
No devices found.						

Step 3 In the **Add Test Device** dialog box displayed, select **Yes**, set the parameters, and click **OK**.

- **Device Name** can contain only letters, digits, and underlines (_) and must be unique in the product.
- **Device Indentity** must be set to a unique value, such as the IMEI or MAC address of the device.
- Choose **Unencrypt** or **Encrypt** based on site requirements.


Add Test Device

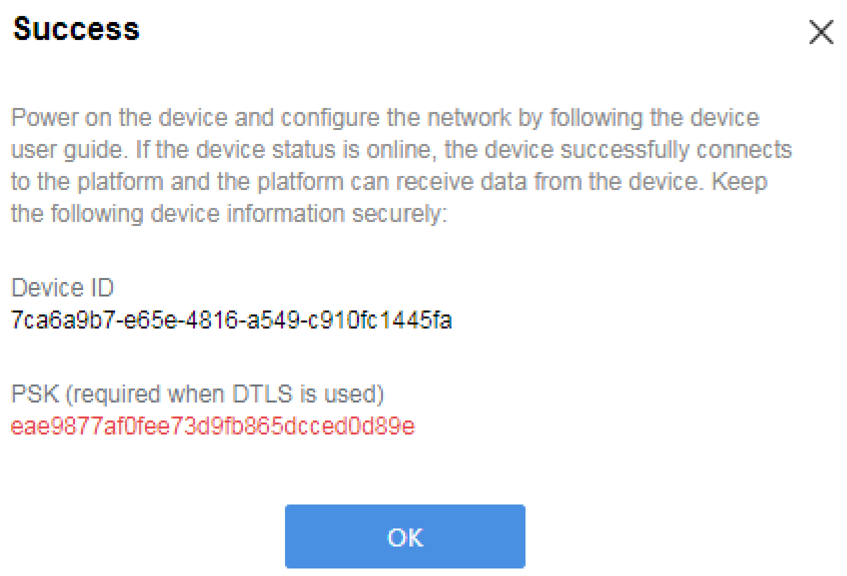
Is Physical Device Available
[✓ Yes] No

*Device Name
testdevice001

*Node ID
111111111

Device Registration Mode
[✓ Unencrypted] Encrypted

After the device is added, **Device ID** and **PSK** are returned. Keep the PSK securely as it is required when the device uses DTLS to connect to the IoT platform.



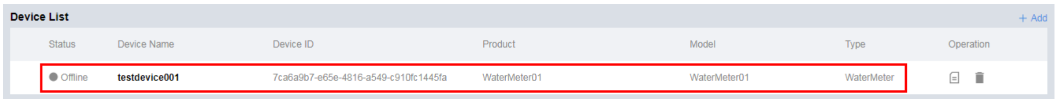
Success

Power on the device and configure the network by following the device user guide. If the device status is online, the device successfully connects to the platform and the platform can receive data from the device. Keep the following device information securely:

Device ID
7ca6a9b7-e65e-4816-a549-c910fc1445fa

PSK (required when DTLS is used)
eae9877af0fee73d9fb865dcced0d89e

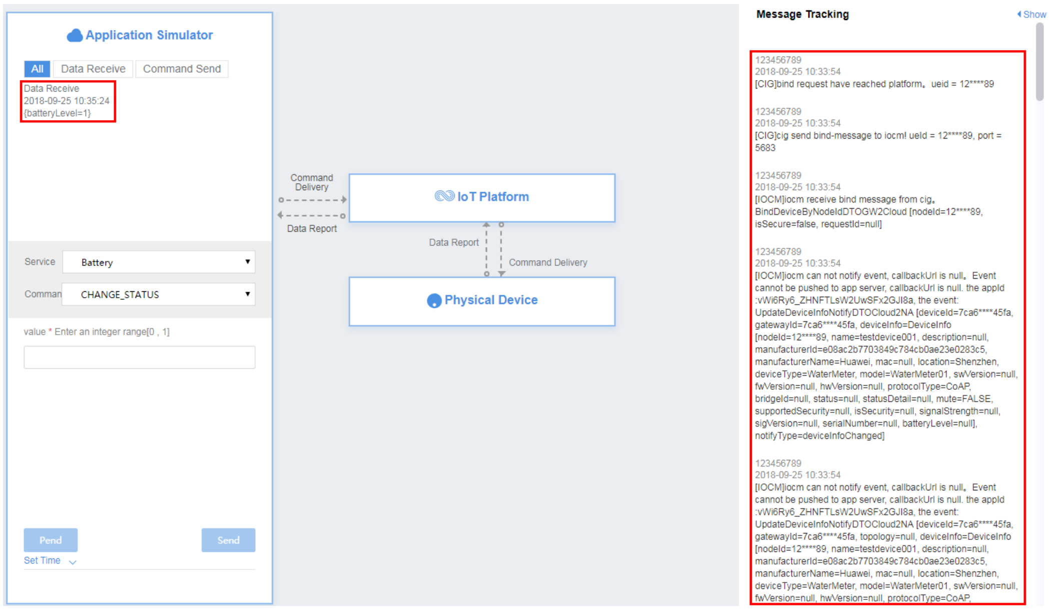
Step 4 In the device list, select the newly added physical device to enter the **Online Test** page.



Device List

Status	Device Name	Device ID	Product	Model	Type	Operation
● Offline	testdevice001	7ca6a9b7-e65e-4816-a549-c910fc1445fa	WaterMeter01	WaterMeter01	WaterMeter	

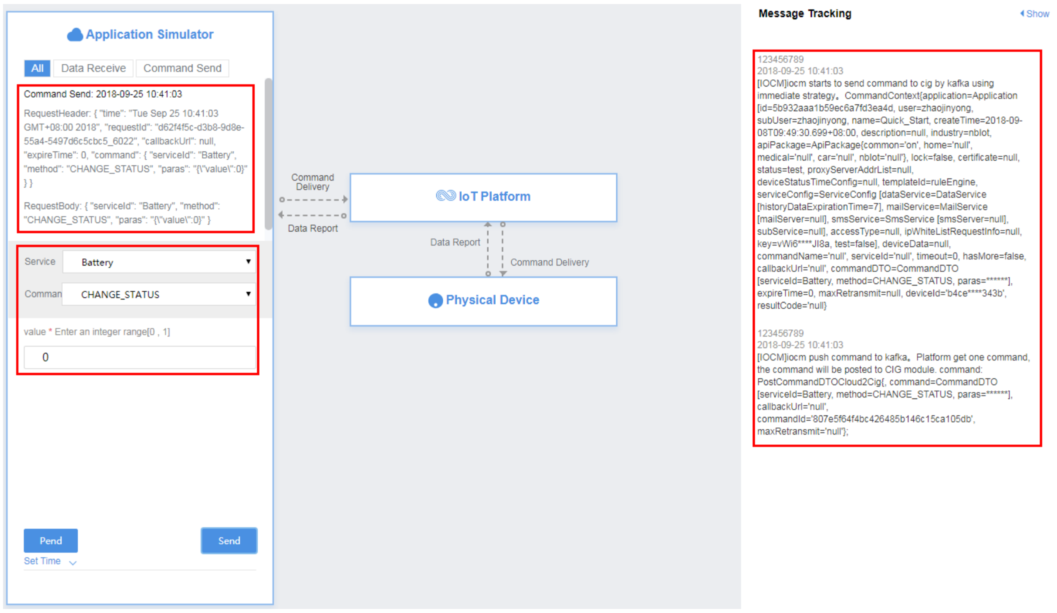
Step 5 Connect the device to the IoT platform and report data. View the data reporting result in **Application Simulator** and processing logs of the IoT platform in **Message Tracking**.



```
Data Receive
2018-09-25 10:35:24
{batteryLevel=1}
```

```
123456789
2018-09-25 10:33:54
[CIG]bind request have reached platform。ueid = 12****89

123456789
2018-09-25 10:33:54
[CIG]cig send bind-message to iocm! ueId = 12****89, port =
5683

123456789
2018-09-25 10:33:54
[IOCM]iocm receive bind message from cig。
BindDeviceByNodeIdDTOGW2Cloud [nodeId=12****89,
isSecure=false, requestId=null]

123456789
2018-09-25 10:33:54
[IOCM]iocm can not notify event, callbackUrl is null。Event
cannot be pushed to app server, callbackUrl is null. the appId
:vWi6Ry6_ZHNFTLsW2UwSFx2GJI8a, the event:
UpdateDeviceInfoNotifyDTOCloud2NA [deviceId=7ca6****45fa,
gatewayId=7ca6****45fa, deviceInfo=DeviceInfo
[nodeId=12****89, name=testdevice001, description=null,
manufacturerId=e08ac2b7703849c784cb0ae23e0283c5,
manufacturerName=Huawei, mac=null, location=Shenzhen,
deviceType=WaterMeter, model=WaterMeter01, swVersion=null,
fwVersion=null, hwVersion=null, protocolType=CoAP,
bridgeId=null, status=null, statusDetail=null, mute=FALSE,
supportedSecurity=null, isSecurity=null, signalStrength=null,
sigVersion=null, serialNumber=null, batteryLevel=null],
notifyType=deviceInfoChanged]

123456789
2018-09-25 10:33:54
[IOCM]iocm can not notify event, callbackUrl is null。Event
cannot be pushed to app server, callbackUrl is null. the appId
:vWi6Ry6_ZHNFTLsW2UwSFx2GJI8a, the event:
UpdateDeviceInfoNotifyDTOCloud2NA [deviceId=7ca6****45fa,
gatewayId=7ca6****45fa, topology=null, deviceInfo=DeviceInfo
[nodeId=12****89, name=testdevice001, description=null,
manufacturerId=e08ac2b7703849c784cb0ae23e0283c5,
manufacturerName=Huawei, mac=null, location=Shenzhen,
deviceType=WaterMeter, model=WaterMeter01, swVersion=null,
fwVersion=null, hwVersion=null, protocolType=CoAP,
```

Step 6 Deliver a command in **Application Simulator**. View processing logs of the IoT platform in **Message Tracking** and check the received command on the device.



Command Send: 2018-09-25 10:41:03

```
RequestHeader: { "time": "Tue Sep 25 10:41:03 GMT+08:00 2018", "requestId": "d62f4f5c-d3b8-9d8e-55a4-5497d6c5cbc5_6022", "callbackUrl": null, "expireTime": 0, "command": { "serviceId": "Battery", "method": "CHANGE_STATUS", "paras": "{\"value\":0}" } }

RequestBody: { "serviceId": "Battery", "method": "CHANGE_STATUS", "paras": "{\"value\":0}" }
```

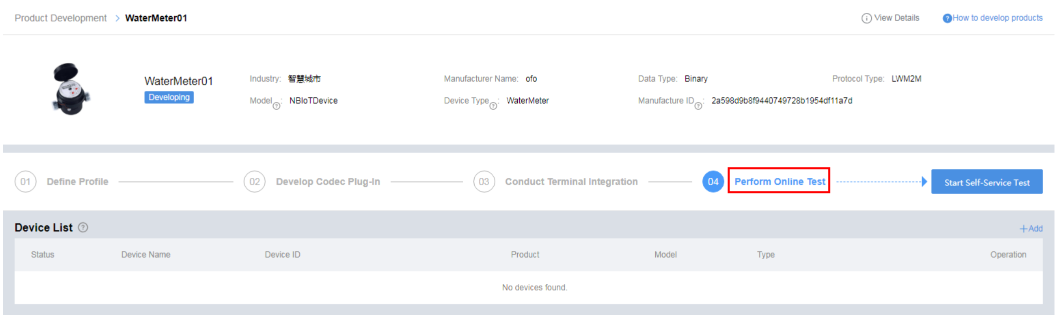
Service: Battery
Comman: CHANGE_STATUS
value * Enter an integer range[0 , 1]: 0

```
123456789
2018-09-25 10:41:03
[IOCM]iocm starts to send command to cig by kafka using immediate strategy。CommandContext{application=Application [id=5b932aaa1b59ec6a7fd3ea4d, user=zhaojinyong, subUser=zhaojinyong, name=Quick_Start, createTime=2018-09-08T09:49:30.699+08:00, description=null, industry=nbIot, apiPackage=ApiPackage{common='on', home='null', medical='null', car='null', nbIot='null'}, lock=false, certificate=null, status=test, proxyServerAddrList=null, deviceStatusTimeConfig=null, templateId=ruleEngine, serviceConfig=ServiceConfig [dataService=DataService [historyDataExpirationTime=7], mailService=MailService [mailServer=null], smsService=SmsService [smsServer=null], subService=null], accessType=null, ipWhiteListRequestInfo=null, key=vWi6****JI8a, test=false], deviceData=null, commandName='null', serviceId='null', timeout=0, hasMore=false, callbackUrl='null', commandDTO=CommandDTO [serviceId=Battery, method=CHANGE_STATUS, paras=******], expireTime=0, maxRetransmit=null, deviceId='b4ce****343b', resultCode='null}

123456789
2018-09-25 10:41:03
[IOCM]iocm push command to kafka。Platform get one command, the command will be posted to CIG module. command: PostCommandDTOCloud2Cig{, command=CommandDTO [serviceId=Battery, method=CHANGE_STATUS, paras=******], callbackUrl='null', commandId='807e5f64f4bc426485b146c15ca105db', maxRetransmit='null'};
```

----End

Using a Virtual Device for Online Test

Step 1 In the product development space, click **Perform Online Test**.

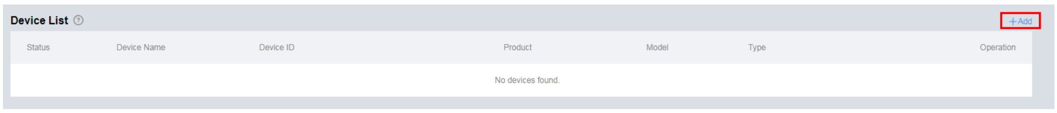


WaterMeter01

Industry:	智慧城市	Manufacturer Name:	ofo	Data Type:	Binary	Protocol Type:	LWM2M
Model:	NBIoTDevice	Device Type:	WaterMeter	Manufacture ID:	2a598d9b8f9440749728b1954df11a7d		

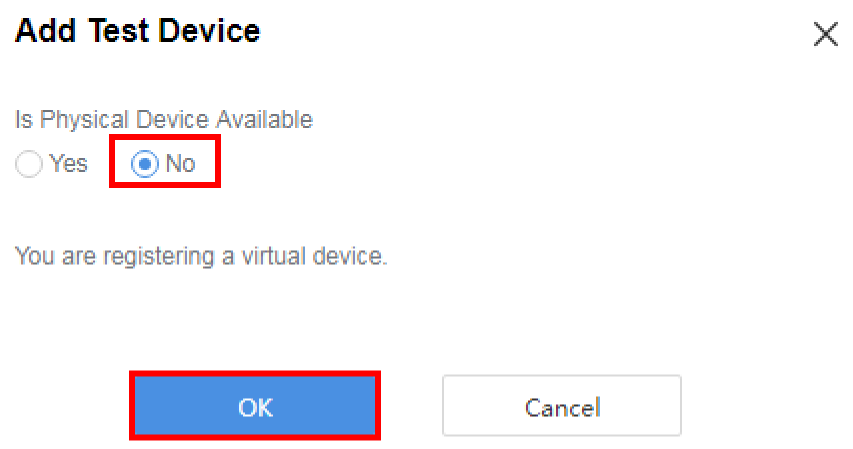
Status	Device Name	Device ID	Product	Model	Type	Operation

Step 2 Click **Add** at the row where **Device List** resides.



Status	Device Name	Device ID	Product	Model	Type	Operation

Step 3 In the **Add Test Device** dialog box displayed, select **No** and click **OK**.

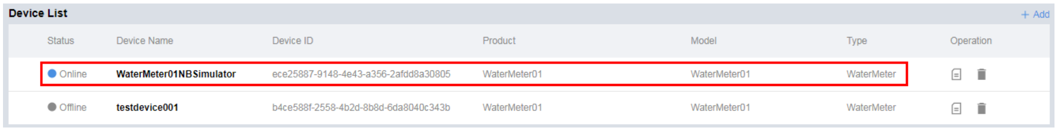

Add Test Device

Is Physical Device Available

Yes [✓ No]

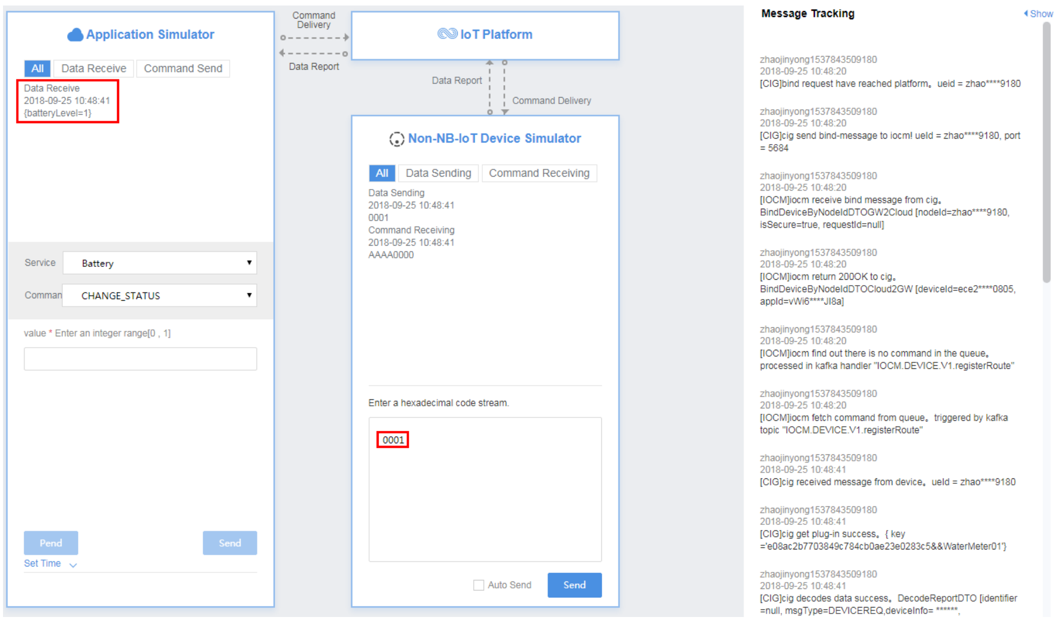
You are registering a virtual device.

Step 4 In the device list, select the newly added virtual device to enter the **Online Test** page. The name of the virtual device is in the format of **Product Name**+**Simulator**. Only one virtual device can be added for each product.


Device List

Status	Device Name	Device ID	Product	Model	Type	Operation
● Online	WaterMeter01NBSimulator	ece25887-9148-4e43-a356-2afdd8a30805	WaterMeter01	WaterMeter01	WaterMeter	
● Offline	testdevice001	b4ce588f-2558-4b2d-8b8d-6da8040c343b	WaterMeter01	WaterMeter01	WaterMeter	

Step 5 In **Device Simulator**, enter a hexadecimal code stream or JSON data (for example, enter a hexadecimal code stream) and click **Send**. Then, view the data reporting result in **Application Simulator** and processing logs of the IoT platform in **Message Tracking**.



Data Receive
2018-09-25 10:48:41
{batteryLevel=1}

Service Battery

Comman CHANGE_STATUS

value * Enter an integer range[0 , 1]

Data Sending
2018-09-25 10:48:41
0001
Command Receiving
2018-09-25 10:48:41
AAAA0000

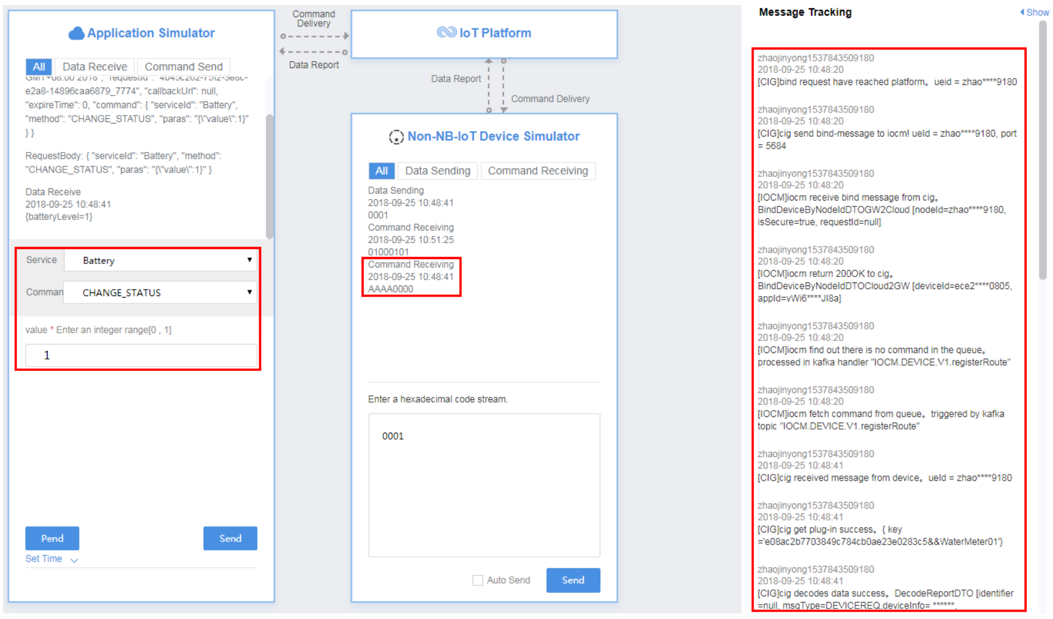
Enter a hexadecimal code stream.

0001

Message Tracking

```
zhaojinyong1537843509180
2018-09-25 10:48:20
[CIG]bind request have reached platform。ueid = zhao****9180

zhaojinyong1537843509180
2018-09-25 10:48:20
[CIG]cig send bind-message to iocm! ueId = zhao****9180, port
= 5684

zhaojinyong1537843509180
2018-09-25 10:48:20
[IOCM]iocm receive bind message from cig。
BindDeviceByNodeIdDTOGW2Cloud [nodeId=zhao****9180,
isSecure=true, requestId=null]

zhaojinyong1537843509180
2018-09-25 10:48:20
[IOCM]iocm return 200OK to cig。
BindDeviceByNodeIdDTOCloud2GW [deviceId=ece2****0805,
appId=vWi6****JI8a]

zhaojinyong1537843509180
2018-09-25 10:48:20
[IOCM]iocm find out there is no command in the queue。
processed in kafka handler "IOCM.DEVICE.V1.registerRoute"

zhaojinyong1537843509180
2018-09-25 10:48:20
[IOCM]iocm fetch command from queue。triggered by kafka
topic "IOCM.DEVICE.V1.registerRoute"

zhaojinyong1537843509180
2018-09-25 10:48:41
[CIG]cig received message from device。ueId = zhao****9180

zhaojinyong1537843509180
2018-09-25 10:48:41
[CIG]cig get plug-in success。{ key
='e08ac2b7703849c784cb0ae23e0283c5&&WaterMeter01'}

zhaojinyong1537843509180
2018-09-25 10:48:41
[CIG]cig decodes data success。DecodeReportDTO [identifier
=null, msgType=DEVICEREQ,deviceInfo= ******,
```

Step 6 Deliver a command in **Application Simulator**. View the received command (for example, a hexadecimal code stream) in **Device Simulator** and processing logs of the IoT platform in **Message Tracking**.



```
e2a8-14896caa6879_7774", "callbackUrl": null,
"expireTime": 0, "command": { "serviceId": "Battery",
"method": "CHANGE_STATUS", "paras": "{\"value\":1}"
} }

RequestBody: { "serviceId": "Battery", "method":
"CHANGE_STATUS", "paras": "{\"value\":1}" }

Data Receive
2018-09-25 10:48:41
{batteryLevel=1}
```

Service: Battery
Comman: CHANGE_STATUS
value * Enter an integer range[0 , 1]: 1

```
Data Sending
2018-09-25 10:48:41
0001
Command Receiving
2018-09-25 10:51:25
01000101
Command Receiving
2018-09-25 10:48:41
AAAA0000
```

Enter a hexadecimal code stream.

```
0001
```

```
zhaojinyong1537843509180
2018-09-25 10:48:20
[CIG]bind request have reached platform。ueid = zhao****9180

zhaojinyong1537843509180
2018-09-25 10:48:20
[CIG]cig send bind-message to iocm! ueId = zhao****9180, port
= 5684

zhaojinyong1537843509180
2018-09-25 10:48:20
[IOCM]iocm receive bind message from cig。
BindDeviceByNodeIdDTOGW2Cloud [nodeId=zhao****9180,
isSecure=true, requestId=null]

zhaojinyong1537843509180
2018-09-25 10:48:20
[IOCM]iocm return 200OK to cig。
BindDeviceByNodeIdDTOCloud2GW [deviceId=ece2****0805,
appId=vWi6****JI8a]

zhaojinyong1537843509180
2018-09-25 10:48:20
[IOCM]iocm find out there is no command in the queue。
processed in kafka handler "IOCM.DEVICE.V1.registerRoute"

zhaojinyong1537843509180
2018-09-25 10:48:20
[IOCM]iocm fetch command from queue。triggered by kafka
topic "IOCM.DEVICE.V1.registerRoute"

zhaojinyong1537843509180
2018-09-25 10:48:41
[CIG]cig received message from device。ueId = zhao****9180

zhaojinyong1537843509180
2018-09-25 10:48:41
[CIG]cig get plug-in success。{ key
='e08ac2b7703849c784cb0ae23e0283c5&&WaterMeter01'}

zhaojinyong1537843509180
2018-09-25 10:48:41
[CIG]cig decodes data success。DecodeReportDTO [identifier
=null, msgType=DEVICEREQ,deviceInfo= ******,
```

----End

1.7 Self-Service Testing

1.7.1 Self-Service Testing Guide

Overview

Self-service testing provides end-to-end test cases to help developers test basic device capabilities, such as data reporting and command delivery. It aims to help you find product defects or problems and shorten the time to market (TTM). After the testing is complete, a test report is generated by the Developer Center for product release certification.

Prerequisites

You have defined the product profile, developed the codec, and deployed the codec.

Procedure

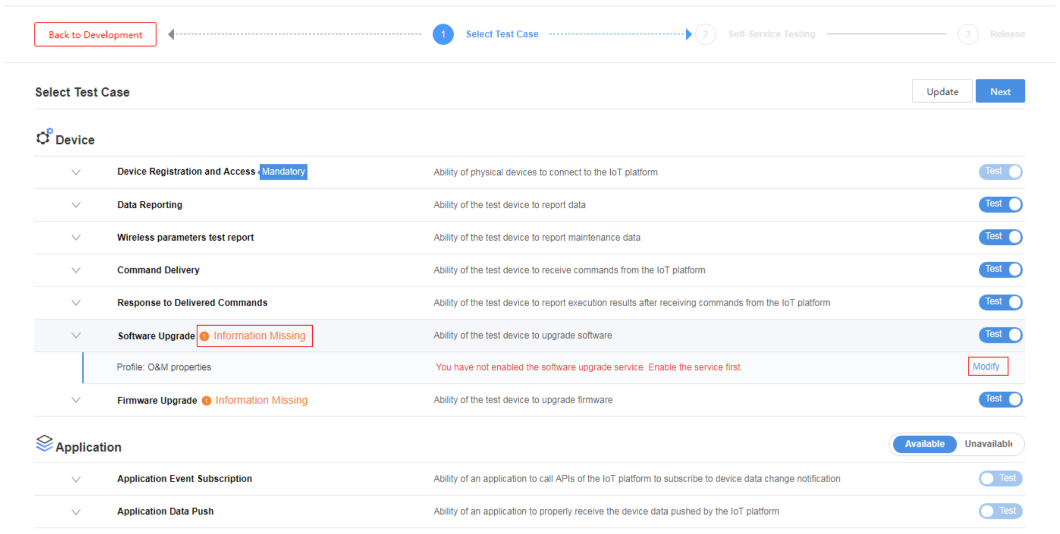
Step 1 In the product development space, click **Self-Service Testing**.

Step 2 The **Select Test Case** page is displayed. You can select test cases as needed. The system automatically checks whether the selected test cases meet the test requirements and returns the check results.

- If all selected test cases pass the check, click **Next** to proceed to the next phase.
- If a test case fails to pass the check, click **Information Missing** on the right of the test case and modify the profile file or codec as prompted.

NOTE

- To initiate the self-service testing, in addition to the mandatory test case, either Data Reporting or Command Delivery must be selected.
- The more cases of a product pass the test, the higher the pass rate of the product release to the Product Center. It is recommended that either Software Upgrade or Firmware Upgrade be selected and all other test cases be included.



Select Test Case

Device

Device Registration and Access Mandatory	Ability of physical devices to connect to the IoT platform	Test
Data Reporting	Ability of the test device to report data	Test
Wireless parameters test report	Ability of the test device to report maintenance data	Test
Command Delivery	Ability of the test device to receive commands from the IoT platform	Test
Response to Delivered Commands	Ability of the test device to report execution results after receiving commands from the IoT platform	Test
Software Upgrade Information Missing	Ability of the test device to upgrade software	Test
Profile: O&M properties	You have not enabled the software upgrade service. Enable the service first.	Modify
Firmware Upgrade Information Missing	Ability of the test device to upgrade firmware	Test

Application Available Unavailable

Application Event Subscription	Ability of an application to call APIs of the IoT platform to subscribe to device data change notification	Test
Application Data Push	Ability of an application to properly receive the device data pushed by the IoT platform	Test

Step 3 Perform the self-service testing as prompted. After the testing is complete, you can preview the test report or apply for releasing the product.

----End

1.7.2 Device Registration and Access Test

Overview

The device registration and access test verifies the capability of the device to connect to the IoT platform. This test includes device registration and device access.

NOTE

This test case is the prerequisite for other tests. If it fails, other tests cannot be performed.

Procedure

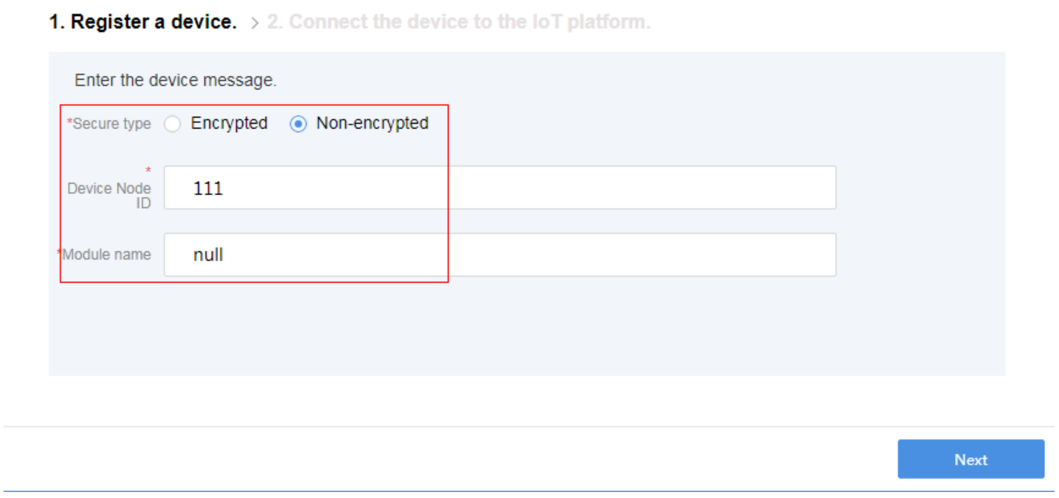
Step 1 On the device registration and access test page, click **Next**. The device registration page is displayed.

Step 2 Enter the test page according to the wizard, select the device type, enter the device node ID and module name, and then click **Next**.

If **Device Type** is set to **Encrypted**, **PSK** needs to be set.

NOTE

If no module is used for the test, set **Module Name** to **Null**.



Enter the device message.

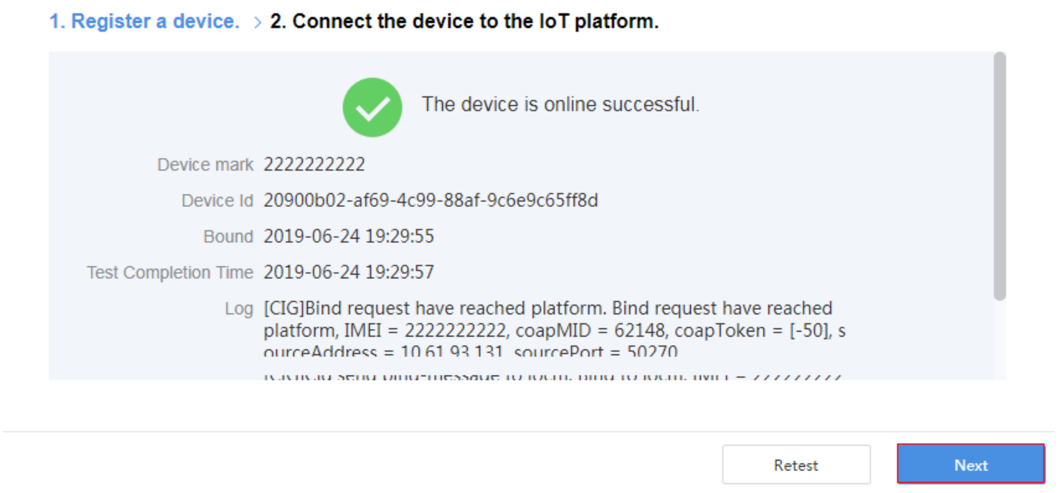
*Secure type ○ Encrypted [✓ Non-encrypted]

*Device Node ID: 111

*Module name: null

Step 3 Register a physical device on the IoT platform according to the wizard, and view the test result.

- If the test is successful, click **Next** to proceed to the next phase.
- If the test fails, rectify the fault and click **Retest**.



The device is online successful.

Device mark	2222222222
Device Id	20900b02-af69-4c99-88af-9c6e9c65ff8d
Bound	2019-06-24 19:29:55
Test Completion Time	2019-06-24 19:29:57
Log	[CIG]Bind request have reached platform. Bind request have reached platform, IMEI = 2222222222, coapMID = 62148, coapToken = [-50], sourceAddress = 10.61.93.131, sourcePort = 50270

----End

1.7.3 Data Reporting Test

Overview

The data reporting test verifies the data reporting capability of a device. The purpose is to test whether the property fields defined in the profile file are correct. If the data format for the IoT platform interacting with the device is binary code stream, the test also verifies whether the mapping between the codec and the profile file is correct.

Procedure

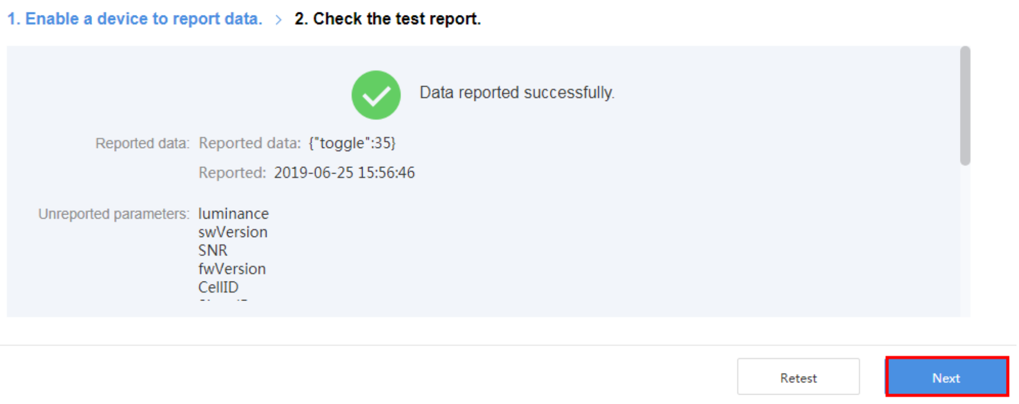
Step 1 On the **Data Reporting** page, click **Next** to start the test.

Step 2 Enter the test page according to the wizard, operate a physical device to report the property data defined in the profile file. If all the data is reported, you can click **Stop Testing** and view the test result.

- If the test is successful, click **Next** to proceed to the next phase.
- If the test fails, rectify the fault and click **Retest**.

NOTE

The platform will verify all attribute data that has been successfully reported and record it in the test report. The repeative attributes will only be recorded once.


1. Enable a device to report data. > 2. Check the test report.

Data reported successfully.

Reported data: Reported data: {"toggle":35}
Reported: 2019-06-25 15:56:46

Unreported parameters: luminance
swVersion
SNR
fwVersion
CellID

----End

1.7.4 Radio Parameter Reporting Test

Overview

This test case checks the radio signal data (signal strength, coverage level, signal to noise ratio, and cell ID) reported by the device.

To execute this test case, you need to define the following radio signal parameters in the profile file and set up the mapping in codec.

Parameter	Type	Description
RSRP/rsrp/ signalStrength/ SignalPower	int	Signal strength. The value ranges from -140 to -40.
ECL/signalECL	int	Coverage level. The value ranges from 0 to 2.
SNR/snr/SINR/sinr/ signalSNR	int	Signal to noise ratio. The value ranges from -20 to 30.
CellID/cellId	int	Cell ID. the value ranges from 0 to 2147483647.

Procedure

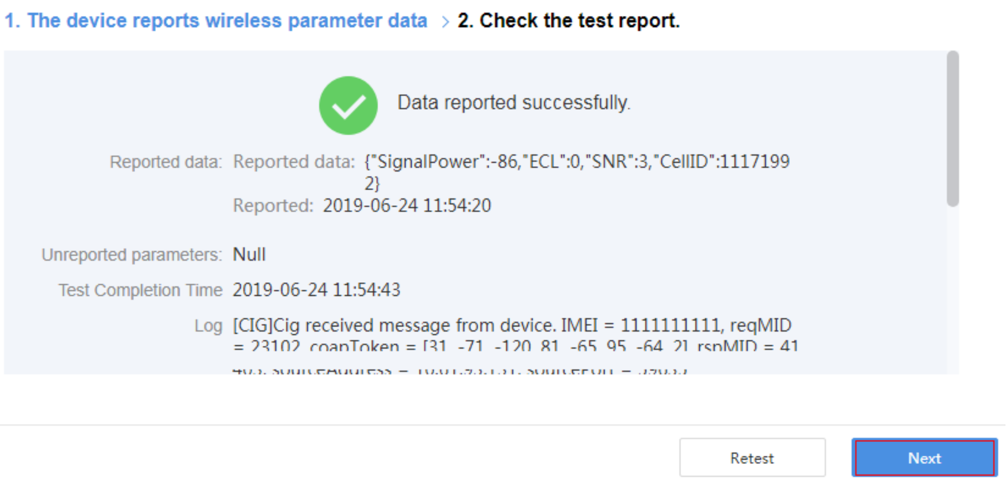
Step 1 On the **Radio Singnal Parameter Reporting** page, click **Next** to start the test.

Step 2 Enter the test page according to the wizard. Use the physical device to report the radio signal parameters defined in the profile file, and view the test result.

- If the test is successful, click **Next** to proceed to the next phase.
- If the test fails, rectify the fault and click **Retest**.

NOTE

The reported radio signal parameter values must be within the ranges.


1. The device reports wireless parameter data
2. Check the test report.
Retest
Next

Data reported successfully.

Reported data: Reported data: {"SignalPower":-86,"ECL":0,"SNR":3,"CellID":11171992}
Reported: 2019-06-24 11:54:20

Unreported parameters: Null

Test Completion Time 2019-06-24 11:54:43

Log [CIG]Cig received message from device. IMEI = 1111111111, reqMID = 23102, coapToken = [31, -71, -120, 81, -65, 95, -64, 21, rspMID = 41

----End

1.7.5 Command Delivery Test

Overview

The command delivery test verifies the capability of a device to receive and process commands. The purpose is to test whether the command fields defined in the profile file are correct. If the data format for the IoT platform interacting with the device is binary code stream, the test also verifies whether the mapping between the codec and the profile file is correct.

If a service application is used for the test, the test also verifies whether the service application can correctly call the **Creating Device Commands** API of the IoT platform to deliver commands to the device.

Procedure

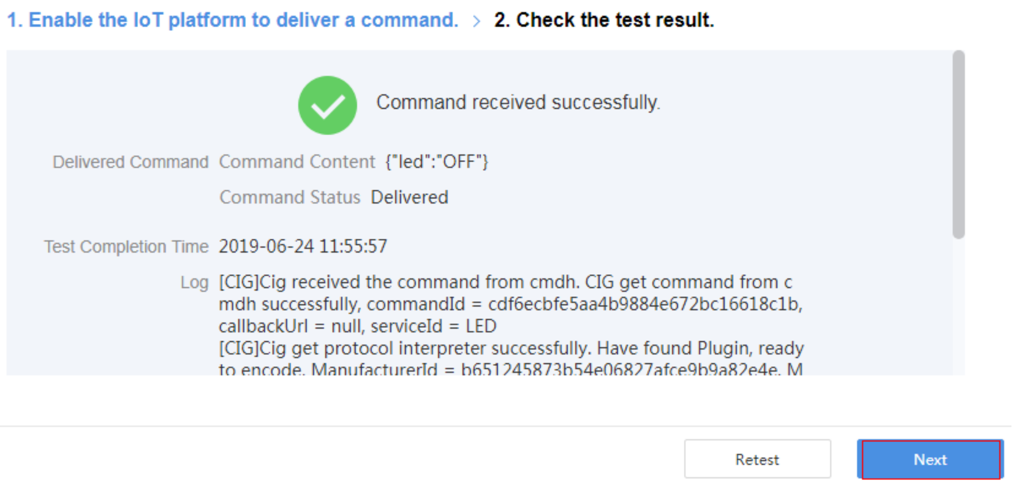
Step 1 On the **Command Delivery** page, click **Next** to start the test.

Step 2 Enter the test page according to the wizard. Deliver a command defined in the profile file to the device on the IoT platform. After the physical device responds to the command, view the test result.

- If the test is successful, click **Next** to proceed to the next phase.
- If the test fails, rectify the fault and click **Retest**.

NOTE

If a service application is connected to the IoT platform, the application server delivers a command to the device. After the physical device responds to the command, the page for **uploading the screenshot about the successful application command delivery** is displayed. Click + on the interface to upload the screenshot. The screenshot is a credential for the service application to correctly call the **Creating Device Commands** API of the IoT platform.


1. Enable the IoT platform to deliver a command.
2. Check the test result.
Retest
Next

Command received successfully.

Delivered Command Command Content {"led":"OFF"}

Command Status Delivered

Test Completion Time 2019-06-24 11:55:57

```
Log [CIG]Cig received the command from cmdh. CIG get command from c
    mdh successfully, commandId = cdf6ecbfe5aa4b9884e672bc16618c1b,
    callbackUrl = null, serviceId = LED
    [CIG]Cig get protocol interpreter successfully. Have found Plugin, ready
    to encode. ManufacturerId = b651245873b54e06827afce9b9a82e4e, M
```

----End

1.7.6 Command Response Test

Overview

The command response test verifies the capability of a device to report the execution result after receiving a command from the IoT platform. When the command delivery response fields have been defined in the profile file (the device is required to return a command execution result), test the command response.

Procedure

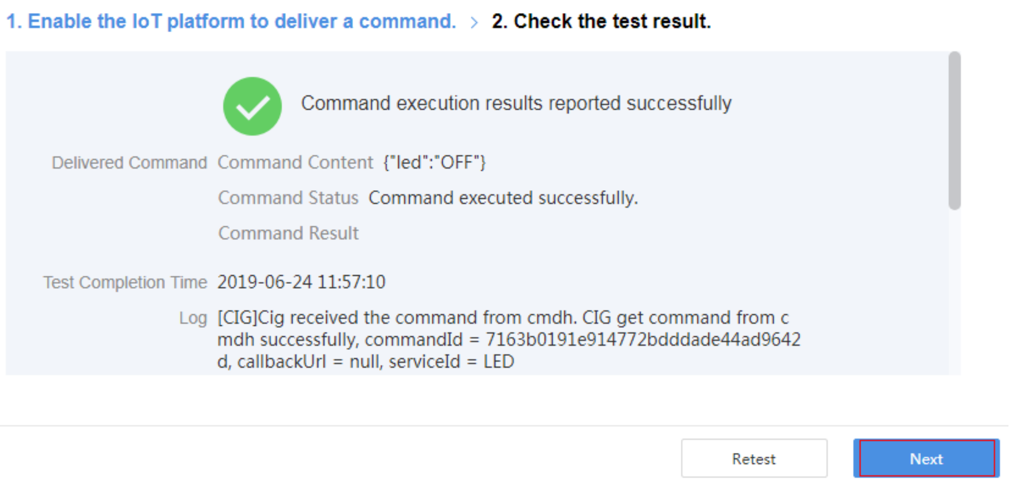
Step 1 On the **Command Response** page, click **Next** to start the test.

Step 2 Enter the test page according to the wizard. Deliver a command to the device on the IoT platform according to the profile file. If the physical device can automatically return a command execution result, you can directly view the test result when it receives the command. If the physical device cannot automatically return a command execution result, manually enable the physical device to report the result based on the command received, and then view the test result.

- If the test is successful, click **Next** to proceed to the next phase.
- If the test fails, rectify the fault and click **Retest**.

NOTE

If a service application is connected to the IoT platform, enable the application server to deliver a command to the device.


1. Enable the IoT platform to deliver a command. > 2. Check the test result.
Retest
Next

Command execution results reported successfully

Delivered Command Command Content {"led":"OFF"}
Command Status Command executed successfully.
Command Result

Test Completion Time 2019-06-24 11:57:10

Log [CIG]Cig received the command from cmdh. CIG get command from cmdh successfully, commandId = 7163b0191e914772bdddade44ad9642d, callbackUrl = null, serviceId = LED

----End

1.7.7 Firmware Upgrade Test

Overview

The firmware upgrade test verifies whether a device supports firmware upgrade. Before performing a firmware upgrade test, ensure that the **Firmware Upgrade** has been enabled under **O&M Service** in the **Profile Definition**.

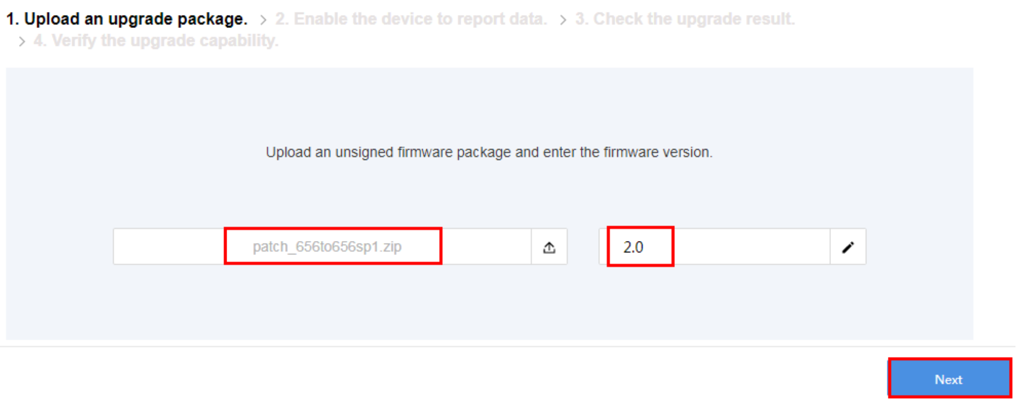
Procedure

Step 1 On the **Firmware Upgrade** page, click **Next** to start the test.

Step 2 Enter the test page according to the wizard, upload the unsigned firmware upgrade package, enter the version number, and click **Next**. The system automatically creates a firmware upgrade task.

NOTE

Ensure that the firmware upgrade package is uploaded and the file is in **.ZIP** format.


1. Upload an upgrade package. > 2. Enable the device to report data. > 3. Check the upgrade result. > 4. Verify the upgrade capability.
Next

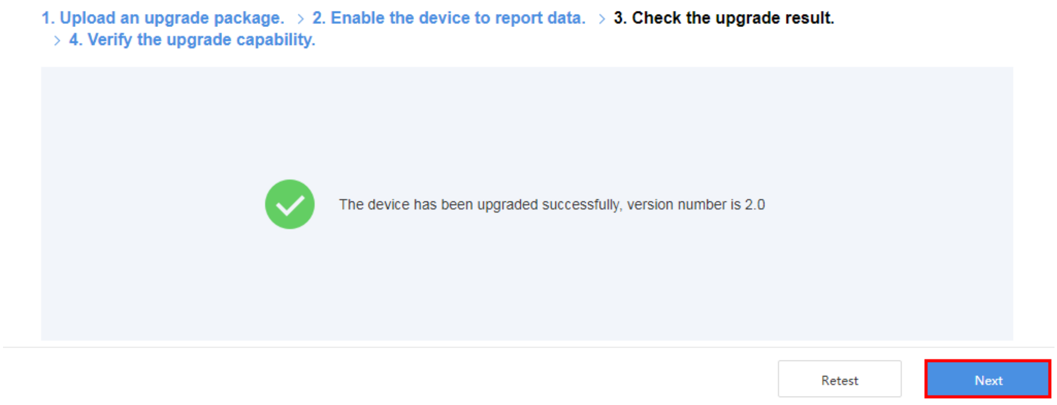
Upload an unsigned firmware package and enter the firmware version.

patch_656to656sp1.zip

2.0

Step 3 Enable a physical device to report property data to trigger the upgrade task. After the upgrade task is complete, view the upgrade result.

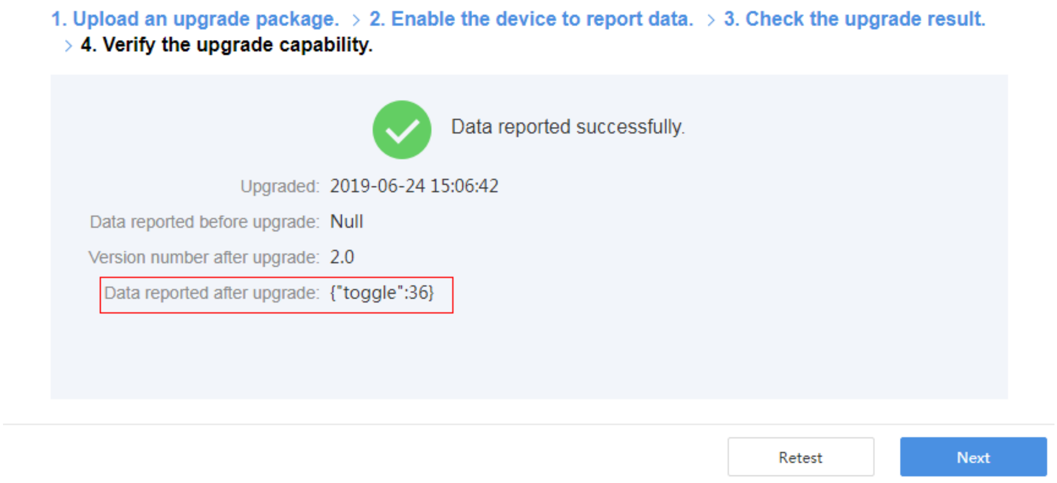
- If the upgrade is successful, click **Next** to check whether the device works properly after the upgrade.
- If the upgrade fails, rectify the fault and click **Retest**.


1. Upload an upgrade package.
2. Enable the device to report data.
3. Check the upgrade result.
4. Verify the upgrade capability.
Retest
Next

The device has been upgraded successfully, version number is 2.0

Step 4 Use the physical device to report property data to check whether the device can communicate with the IoT platform after the upgrade. View the test result.

- If the test is successful, click **Next** to proceed to the next phase.
- If the test fails, rectify the fault and click **Retest**.


1. Upload an upgrade package.
2. Enable the device to report data.
3. Check the upgrade result.
4. Verify the upgrade capability.
Retest
Next

Data reported successfully.

Upgraded:	2019-06-24 15:06:42
Data reported before upgrade:	Null
Version number after upgrade:	2.0
Data reported after upgrade:	{"toggle":36}

----End

1.7.8 Software Upgrade Test

Overview

The software upgrade test verifies whether a device supports software upgrade. Before performing a software upgrade test, ensure that **Software Upgrade** has been enabled under **O&M Service** in the **Profile Definition**.

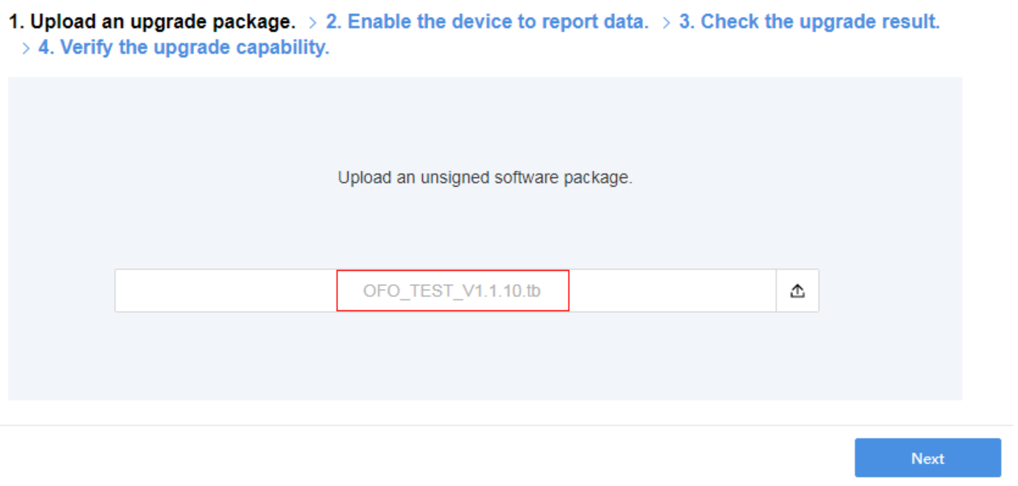
Procedure

Step 1 On the **Software Upgrade** page, click **Next** to start the test.

Step 2 Enter the test page according to the wizard, upload the unsigned software upgrade package, and click **Next**. The system automatically creates a software upgrade task.

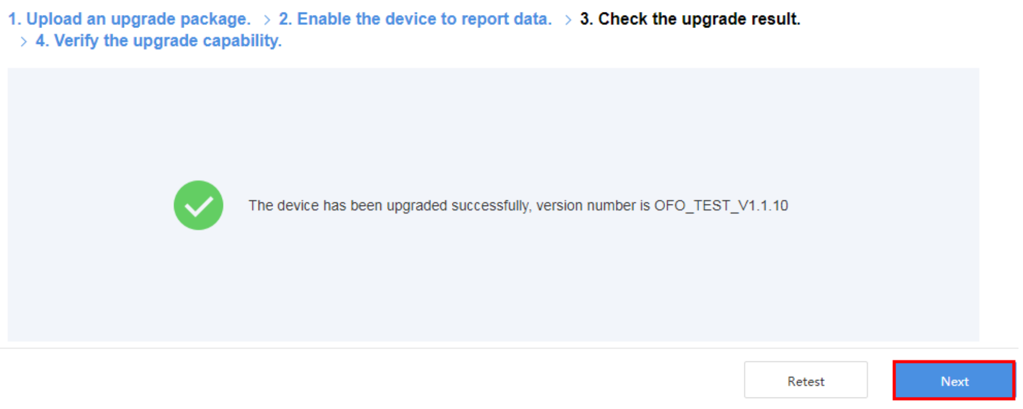
NOTE

Ensure that the software upgrade package is uploaded and the file is in **.ZIP** format.


1. Upload an upgrade package. > 2. Enable the device to report data. > 3. Check the upgrade result. > 4. Verify the upgrade capability.
Upload an unsigned software package.
OFO_TEST_V1.1.10.tb
Next

Step 3 Enable the physical device to report property data to trigger the upgrade task. View the test result.

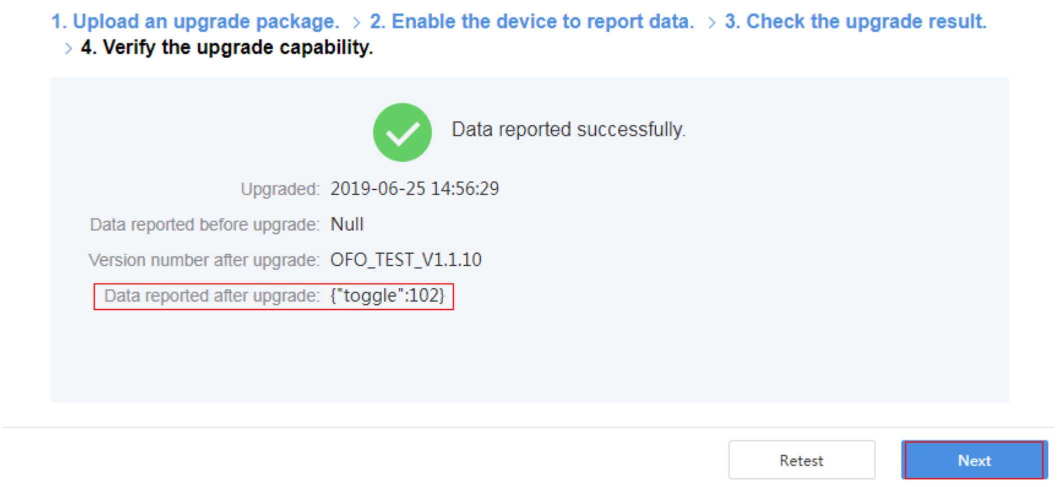
- If the upgrade is successful, click **Next** to check whether the device works properly after the upgrade.
- If the upgrade fails, rectify the fault and click **Retest**.


1. Upload an upgrade package. > 2. Enable the device to report data. > 3. Check the upgrade result. > 4. Verify the upgrade capability.
Retest
Next

The device has been upgraded successfully, version number is OFO_TEST_V1.1.10

Step 4 Enable the physical device to report property data to check whether the device can communicate with the IoT platform after the upgrade. Check the test result.

- If the test is successful, click **Next** to proceed to the next phase.
- If the test fails, rectify the fault and click **Retest**.


1. Upload an upgrade package. > 2. Enable the device to report data. > 3. Check the upgrade result. > 4. Verify the upgrade capability.
Retest
Next

Data reported successfully.

Upgraded: 2019-06-25 14:56:29
Data reported before upgrade: Null
Version number after upgrade: OFO_TEST_V1.1.10
Data reported after upgrade: {"toggle":102}

----End

1.7.9 Application Subscription Event Test

Overview

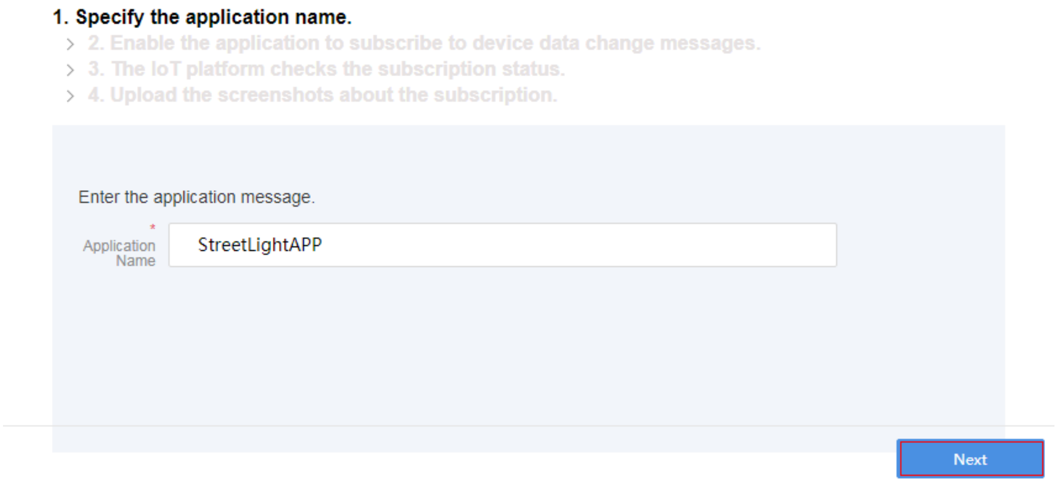
The application subscription event test verifies whether the service application can correctly call the **Subscribing to Service Data of the IoT Platform** API to subscribe to the device data changes.

If the IoT platform uses HTTPS to push data to a service application, the CA certificate provided by the service application must be uploaded to the IoT platform. To load a CA certificate, choose **Applications** > **Interconnection**, click **Certificate Management**, and **Add** in the **Push Certificate** area. For details, see **How Do I Export the HTTPS Push Certificate**.

Procedure

Step 1 On the application subscription event test page, click **Next**.

Step 2 Enter the application name and click **Next**.


1. Specify the application name.
> 2. Enable the application to subscribe to device data change messages.
> 3. The IoT platform checks the subscription status.
> 4. Upload the screenshots about the subscription.
Next

Enter the application message.

* Application Name: StreetLightAPP

Step 3 Enter the test page according to the wizard, subscribe to device data change messages, and view the test result.

- The message subscription is successful. Click **Next** to upload the screenshot about the successful subscription.
- If the message subscription fails, rectify the fault and click **Retest**.

Step 4 Click + on the **Upload the screenshots about the subscription** page and upload the screenshot. This screenshot is a credential for the service application to correctly call the API for **Subscribing to Service Data of the IoT Platform** API on the IoT platform.

After the screenshot is uploaded, click **Next** to proceed to the next phase.

NOTE

The size of the screenshot to be uploaded cannot exceed 20 MB.

----End

1.7.10 Application Data Push Test

Overview

This test verifies whether the service application can correctly receive data pushed by the IoT platform.

Procedure

Step 1 On the application data push test page, click **Next**.

Step 2 Enter the test page according to the wizard. Use the physical device to report property data defined in the profile file. The IoT platform obtains the data and pushes it to the service application. View the test result.

- If the data is pushed successfully, click **Next** to upload the screenshot about the application ceceiving the data.
- If the test fails, rectify the fault and click **Retest**.

Step 3 Click + on the **Upload the screenshot about the application receiving the data** page and upload the screenshot. This screenshot is a credential for the service application to correctly receive data pushed by the IoT platform.

NOTE

The size of the screenshot to be uploaded cannot exceed 20 MB.

----End

1.8 Product Release

Overview

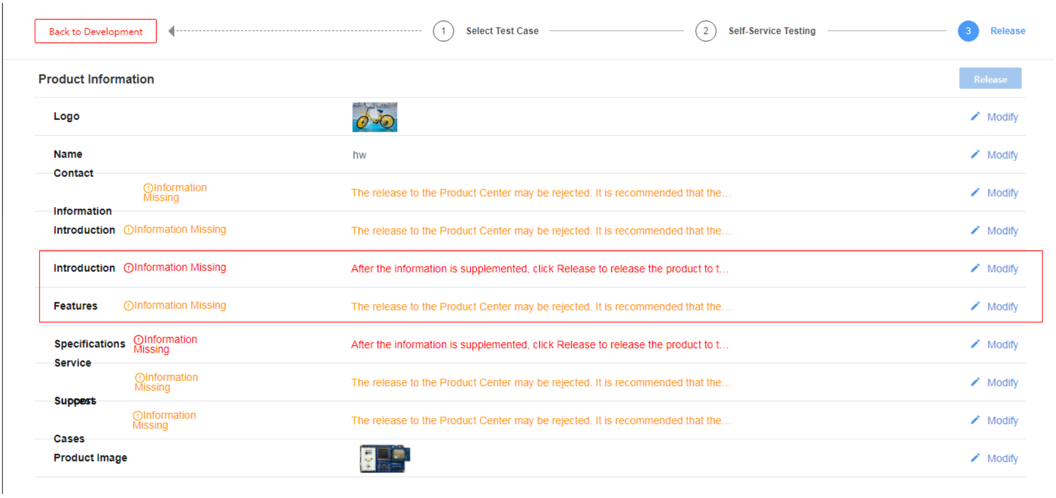
If the Developer Center has interconnected with the Product Center, you can apply to the Product Center for product release. You can release your product and display it in the Product Center or set it visible only to yourself.

Applying for Product Release

Step 1 Click **Apply for Release** after the product passes the test cases.

Step 2 The system automatically checks the integrity of the manufacturer and product information. If no important information is missing, click **Release**.

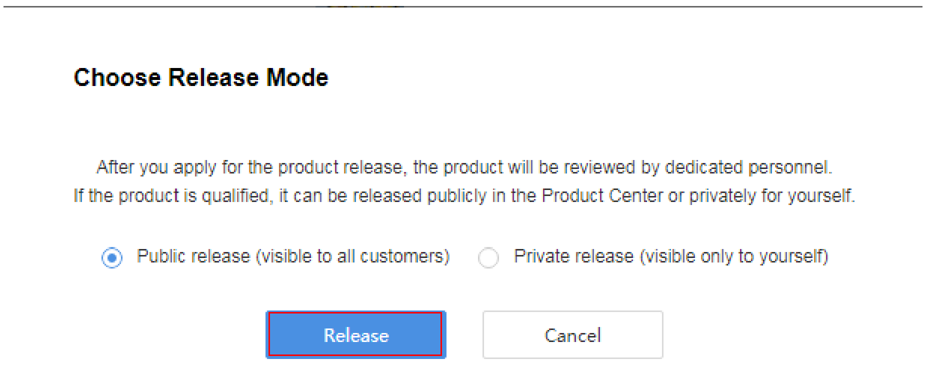
- Information missing in yellow: Some information is incomplete, which does not affect the product release. However, the product may fail to be approved for release in the Product Center. It is recommended that the information be supplemented.
- Information missing in red: Important information is missing. The product can be released only after the information is supplemented.



Product Information Release

Logo			Modify
Name		hw	Modify
Contact	Information Missing	The release to the Product Center may be rejected. It is recommended that the...	Modify
Information Introduction	Information Missing	The release to the Product Center may be rejected. It is recommended that the...	Modify
Introduction	Information Missing	After the information is supplemented, click Release to release the product to t...	Modify
Features	Information Missing	The release to the Product Center may be rejected. It is recommended that the...	Modify
Specifications	Information Missing	After the information is supplemented, click Release to release the product to t...	Modify
Service	Information Missing	The release to the Product Center may be rejected. It is recommended that the...	Modify
Suppests	Information Missing	The release to the Product Center may be rejected. It is recommended that the...	Modify
Cases Product Image			Modify

Step 3 Select a release mode and click **Release**.



Choose Release Mode

After you apply for the product release, the product will be reviewed by dedicated personnel.
If the product is qualified, it can be released publicly in the Product Center or privately for yourself.

[✓ Public release (visible to all customers)] Private release (visible only to yourself)

----End

2 Device Interconnection

2.1 Creating an Application

Overview

Create an application on the Management Portal to connect physical devices and NAs to the IoT platform for device data collection and device management.

After an application is created, the IoT platform assigns the application and device access addresses and ports to support fast NA and device access.

Procedure

Step 1 Log in to the HUAWEI CLOUD management console. Click **IoT Device Management**, and click **Management Portal**.

Step 2 Choose **Application List**, and click **Create Application**.

Step 3 Set the parameters based on **Table 2-1**.

Table 2-1 Application creation parameters

Parameter	Description
Basic Information	
Application Name	Specify the name of an application. It must be unique under the user and cannot be changed.
Industry	Select a value based on the industry attributes of the application.

Parameter	Description
Message Tracing Authorization	Specify whether the IoT platform operations administrator can trace faulty devices. ● If message tracing authorization is enabled, the IoT platform operations administrator, when helping you locate faults, can trace service data reported by devices. When authorization is enabled, **Authorization Validity** must also be specified. The value of **Authorization Validity** can be set to **Custom** or **Always**. To ensure user data rights, the IoT platform operations administrator can retain the device data for a maximum of three days. ● If message tracing authorization is disabled, the IoT platform operations administrator cannot trace service data reported by devices. This may reduce fault locating efficiency. You are advised to enable authorization.
Message Push	
Protocol Selection	**Push Protocol** The push protocol is determined by the transport protocol set when a network application (NA) subscribes to device information from the IoT platform. If the transmission channel for data push is set to HTTP on the NA, you can use HTTPS or HTTP to transmit data. ● **HTTPS**: Encrypted transmission is used between the IoT platform and NA. A CA certificate must be uploaded to the NA. ● **HTTP**: Non-encrypted transmission is used between the IoT platform and NA. This mode is relatively less secure, and data sent between the IoT platform and NA may be disclosed. **CA Certificate** The CA certificate is provided by the NA and used by the IoT platform to verify the NA. **NOTE** The CA certificate preconfigured on the IoT platform is used only for commissioning. In commercial scenarios, use the CA certificate provided by the NA.
Platform Capability	
Device Data Management	The IoT platform can store historical device data. You can enable or disable the storage function. The default value is **On**. ● If the value is **On**, the IoT platform stores historical data. The storage duration is subject to that displayed. ● If the value is **Off**, the IoT platform does not store historical data.
Push Service	The NA subscribes to device information from the IoT platform, and the IoT platform pushes messages to the NA.
Other	
Description	Describe the application.
Application Icon	Specify the icon of the application.

Step 4 Select **I have read and agree to the Terms of Personal Data Use**, and click **Confirm**. After the application is created, the **Success** dialog box is displayed, showing basic information about the application, including the application ID, application secret, application access address, and device access address.

- Click **Save Secret to Local** to save the application secret. The secret is invisible on the application details page. Keep it secure. If you forget the secret, click ⚙ and choose **Reset Secret**. Alternatively, you can open the application details page, click the **Information** tab page, and click **Reset** under **Security**.

  NOTE

  The application ID and application secret are used by the NA to connect to the IoT platform. If you reset the secret, the old secret becomes invalid, and the NA server must use the new secret to access the IoT platform. Exercise caution when performing this operation.

- Click **Go to Application Details** to view the application details page.
- Click **Return to Application List** to display the page for creating an application. Click the application icon to view its details.

Step 5 (Optional) For an NB-IoT device, click the created application. On the **Service Settings** tab page of the application details page, set the working mode of the NB-IoT device. The working mode corresponds to the cache mode of commands delivered by the IoT platform. The working mode must be the same as the working mode used by the device.

- Pending delivery: Set the working mode to PSM. The value of **expireTime** in the command delivery API is used. If **expireTime** is not set, the default time is 48 hours.
- Immediately delivery: Set the working mode to DRX or eDRX. Commands are not cached and are delivered directly.

----End

2.2 Importing a Product Model

Overview

A product model (or profile file) describes the capabilities and features of a device. You can construct an abstract model of a device type by defining a profile file on the IoT platform, allowing it to understand the services, properties, and commands supported by the device.

After a product model is developed and released on the Product Center, import it on the Management Portal.

Procedure

Step 1 Choose **Product Models**, and click **Add**.

Step 2 Import the product model from the Product Center or local PC.

- Import from the Product Center.
  a. Choose **Import from Product Center** to open the **Product Center** page.
  b. Search for a product by product name, device type, or manufacturer name. In the search result, click the name of the product to be imported.

c. Check whether the product is a public product.
   - For a public product, you can click **Import** to import the product model from the Product Center to the IoT platform.
   - For a private product, you must enter the verification code obtained from the Product Center. If the verification is successful, you can view the product details and import the product model to the IoT platform.

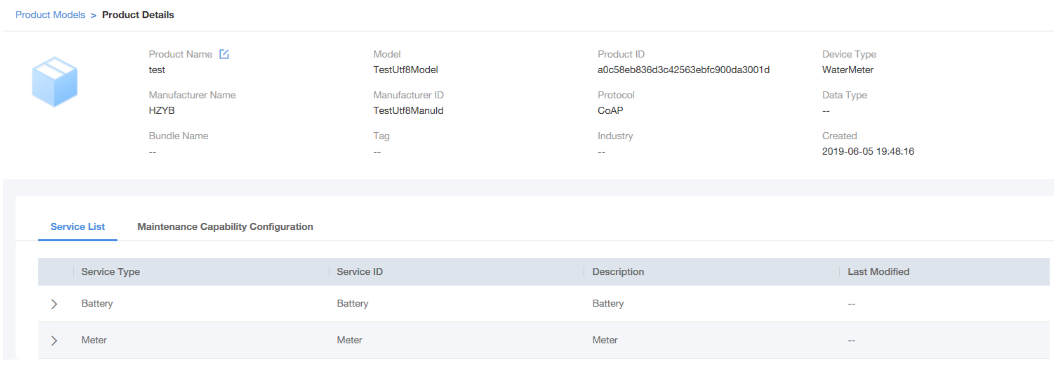
- Import from your local PC.
  a. Choose **Import from Local**.
  b. In the dialog box displayed, enter the product name and upload the resource file.
  c. Click **Confirm** and wait until the import is complete.

  NOTE

  The product ID and product key are used for device registration. Click **Save to Local** to save the product key. The product key is not displayed on the product model details page. Keep it secure.

Step 3 View the import result on the **Product Models** page.

- Import failure: You can view the cause of the import failure in the **Failure Cause** area. This helps with fault locating.
- Import success: You can click **Details** to view product model details.


Product Models > Product Details
Service List
Maintenance Capability Configuration

Product Name	Model	Product ID	Device Type
test	TestUtf8Model	a0c58eb836d3c42563ebfc900da3001d	WaterMeter
Manufacturer Name	Manufacturer ID	Protocol	Data Type
HZYB	TestUtf8ManuId	CoAP	--
Bundle Name	Tag	Industry	Created
--	--	--	2019-06-05 19:48:16

Service Type	Service ID	Description	Last Modified
Battery	Battery	Battery	--
Meter	Meter	Meter	--

NOTE

You can delete a disused product from the product list by clicking **Delete**. After deletion, the devices of this product cannot be used. The functions of the devices under the product are restored only after the product is imported to the Product Center again.

----End

2.3 Registering a Device

Overview

Register a device on the IoT platform and define device parameters. Then the device can connect to the IoT platform if authentication succeeds.

Procedure

Step 1 Choose **Devices** > **Registration**.

Step 2 Click the **Individual Registration** tab, and then click **Register**. In the dialog box displayed, set the parameters based on **Table 2-2**, and click **Confirm**.

Table 2-2 Individual device registration parameters

Parameter	Configuration Rule
Product	Select a product. You can select a product only after it is defined on the **Product Models** page. If the product model has not been uploaded, upload or create it first.
Node ID	Specify the unique physical identifier of a device, such as its IMEI or MAC address. This parameter is carried during device access and used by the IoT platform to authenticate the device. ● For a native MQTT device, the device ID (corresponding to the node ID) and secret generated after the registration are used for IoT platform connection. ● For an NB-IoT device or a device integrated with the AgentLite SDK, the node ID and pre-secret entered during the registration are used for IoT platform connection.
Pre-secret	● For an NB-IoT device, the pre-secret is used to encrypt the transmission channel between it and the IoT platform. ● For a device integrated with the AgentLite SDK, the pre-secret is used by the IoT platform to authenticate its access. ● A native MQTT device does not require a pre-secret.
Confirm Pre-secret	Enter the pre-secret again.

----End

2.4 Connecting a Device

Overview

Connect a physical device to the IoT platform to verify that the device can report data to the IoT platform and display the data on the Management Portal.

Prerequisites

A device has been developed. For details, see **Developing a Device**.

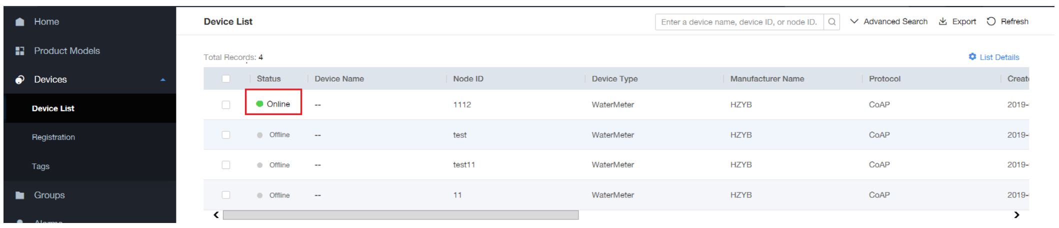
Procedure

Step 1 Set the IoT platform IP address and port number on the device to the device interconnection information of IoT Device Management. You can view the interconnection information on the IoT Management Console.

Step 2 (Optional) If the device is an MQTT device, load the commercial CA certificate provided by the IoT platform to the device.

Step 3 Power on the device to report data to the IoT platform.

Step 4 Log in to the Management Portal. Choose **Device Management** > **Devices** > **Device List**, and check the device status on the device list. If the status is **Online**, the device has been connected to the IoT platform.



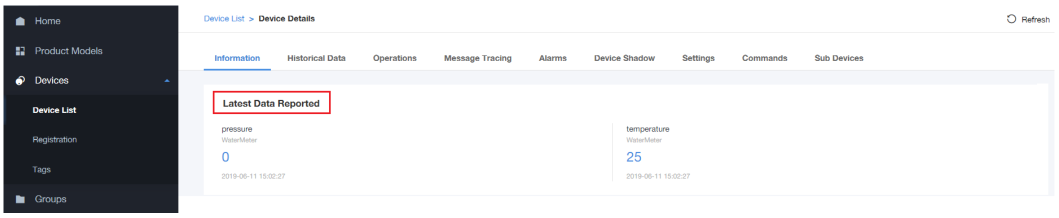
Total Records: 4

	Status	Device Name	Node ID	Device Type	Manufacturer Name	Protocol	Creat
	Online	--	1112	WaterMeter	HZYB	CoAP	2019-
	Offline	--	test	WaterMeter	HZYB	CoAP	2019-
	Offline	--	test11	WaterMeter	HZYB	CoAP	2019-
	Offline	--	11	WaterMeter	HZYB	CoAP	2019-

Step 5 Click the device. On the details page, view the latest reported data. If the data can be properly parsed and displayed, the device reports data successfully.

NOTE

To view all reported historical data, click the **Historical Data** tab.



pressure
WaterMeter
0
2019-06-11 15:02:27

temperature
WaterMeter
25
2019-06-11 15:02:27

Step 6 On the **Commands** tab page of the device details page, click **Send Command**, select a command, and issue the command to the device. Check the execution result. If the device acts as instructed and the execution result of the command delivery task is displayed as **Delivered** or **Successful** on the Management Portal, the device command is delivered successfully.

NOTE

- If the NB-IoT device uses the pending delivery mode, the command is delivered to the device only after the device reports data.
- If the device returns the command execution result (success or failure) to the IoT platform, the task status is updated to **Successful** or **Failed** based on the execution result.

----End

3 Application Interconnection

3.1 Connecting an NA

Overview

Connect an NA to the IoT platform to allow remote device management.

Prerequisites

- An application has been developed. For details, see **Developing an Application**.
- An application has been created. For details, see **Creating an Application**.

Procedure

Step 1 Set the IoT platform IP address and port number on the NA to the application interconnection information of IoT Device Management. You can view the interconnection information on the IoT Management Console.

Step 2 Replace the application ID and application secret on the NA with those allocated in **Creating an Application**.

Step 3 If the NA uses HTTPS to communicate with the IoT platform, replace the commissioning certificate with a commercial certificate.

The unidirectional authentication mode is used when the NA connects to the IoT platform. Therefore, obtain the CA certificate of the IoT platform and load it to the NA.

Step 4 The NA calls the authentication API of the IoT platform to complete the access. For details on the authentication APIs, see the *Northbound API Reference*.

----End

3.2 Subscribing to Data

Overview

An NA calls the subscription API of the IoT platform to inform the IoT platform where a notification is to be pushed and the type of the notification to be pushed, such as device service data and device alarms.

HTTPS is used for subscription push, and the certificate of the NA must be loaded.

Procedure

Step 1 An NA calls the subscription API of the IoT platform to subscribe to data. For details about the subscription API, see the *Northbound API Reference*.

Step 2 Log in to the Management Portal, and choose **System Management** > **Application Management** > **Application List**. On the page displayed, click the created application.

Step 3 On the **Information** tab page of the application details page, click **Manage Certificate** in the **Message Push** pane.

Step 4 Click **Add**. Set the parameters based on **Table 3-1**, and click **Confirm**.

Table 3-1 CA Certificate dialog box

Parameter	Description
CA Certificate	You must apply for and purchase a CA certificate in advance. The CA certificate is provided by the NA.
Domain/IP and Port	Specify the domain name or IP address and port number used by the IoT platform to push messages to the NA. Set this parameter to the domain name or IP address and port number in the callback URL in the subscription API. Example values are **api.ct10649.com:9001** and **127.0.1.2:8080**.
LoadBalance Nickname	Nickname of the LoadBalance to which the certificate is loaded. Retain the default value **Default**.
Check Common Name	Specify whether the common name of the CA certificate is verified to ensure that 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 set to **ON**. This parameter specifies the common name of the CA certificate. Obtain the value from the certificate applicant.

Parameter	Description
Use Device Certificate	Enable this function if the NA needs to verify the validity of the IoT platform. ● If this function is enabled, one-way authentication is used (the IoT platform verifies the server corresponding to callback URL). In this case, unidirectional authentication must also be configured on the server. ● If this function is turned on, the server corresponding to the callback URL must apply for the corresponding certificate file and upload the device certificate on the IoT platform.
Device Certificate	A device certificate, also called public key certificate, is a digital certificate that contains a public key. The device certificate is provided by the NA.
Private Key File	Specify the private key file contained in the user key pair. You can set a password to protect a private key file, preventing access by anyone without the password.
Private Key Password	Specify the password used to encrypt a private key file.

----End

3.3 Commissioning an NA

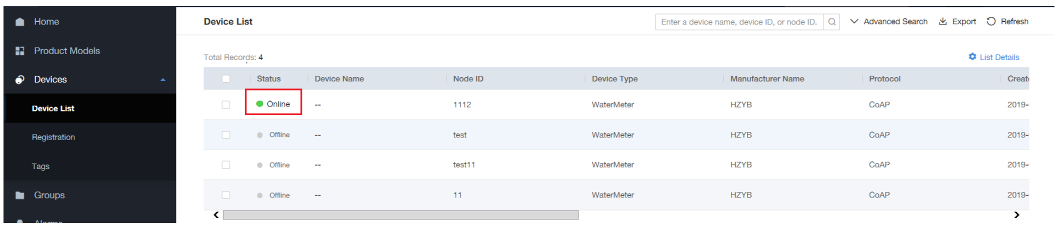
Overview

After connecting a device and an NA to the IoT platform, verify that the NA can receive data reported by the device and that the device can receive and execute commands delivered by the NA.

Procedure

Step 1 Power on the device to report data to the IoT platform.

Step 2 Log in to the Management Portal. Choose **Device Management** > **Devices** > **Device List**, and check the device status on the device list. If the status is **Online**, the device has been connected to the IoT platform.



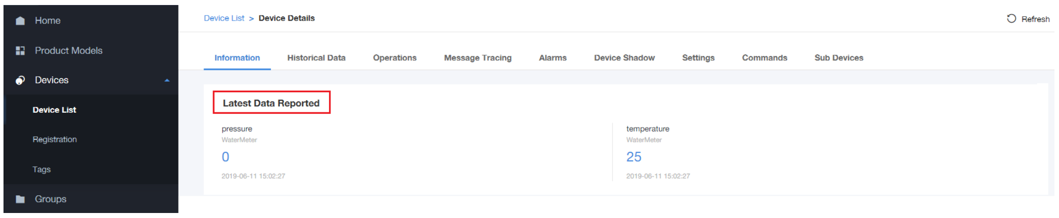
Total Records: 4

	Status	Device Name	Node ID	Device Type	Manufacturer Name	Protocol	Creat
☐	● Online	--	1112	WaterMeter	HZYB	CoAP	2019-
☐	● Offline	--	test	WaterMeter	HZYB	CoAP	2019-
☐	● Offline	--	test11	WaterMeter	HZYB	CoAP	2019-
☐	● Offline	--	11	WaterMeter	HZYB	CoAP	2019-

Step 3 Click the device. On the details page, view the latest reported data. If the data can be properly parsed and displayed, the device reports data successfully.

NOTE

To view all reported historical data, click the **Historical Data** tab.



Step 4 On the server corresponding to callback URL, check whether it has received data from the IoT platform. If so, the IoT platform has successfully pushed the message.

Step 5 Enable the NA to issue a command to the device. Check the execution result. If the device acts as instructed and the execution result of the command delivery task on the Management Portal is displayed as **Delivered** or **Successful**, the NA successfully delivers the command to the device.

NOTE

- If the NB-IoT device uses the pending delivery mode, the command is delivered to the device only after the device reports data.
- If the device returns the command execution result (success or failure) to the IoT platform, the task status is updated to **Successful** or **Failed** based on the execution result.

----End

4 SDK Usage Guide on the Device Side

4.1 LiteOS SDK Integration Development Guide

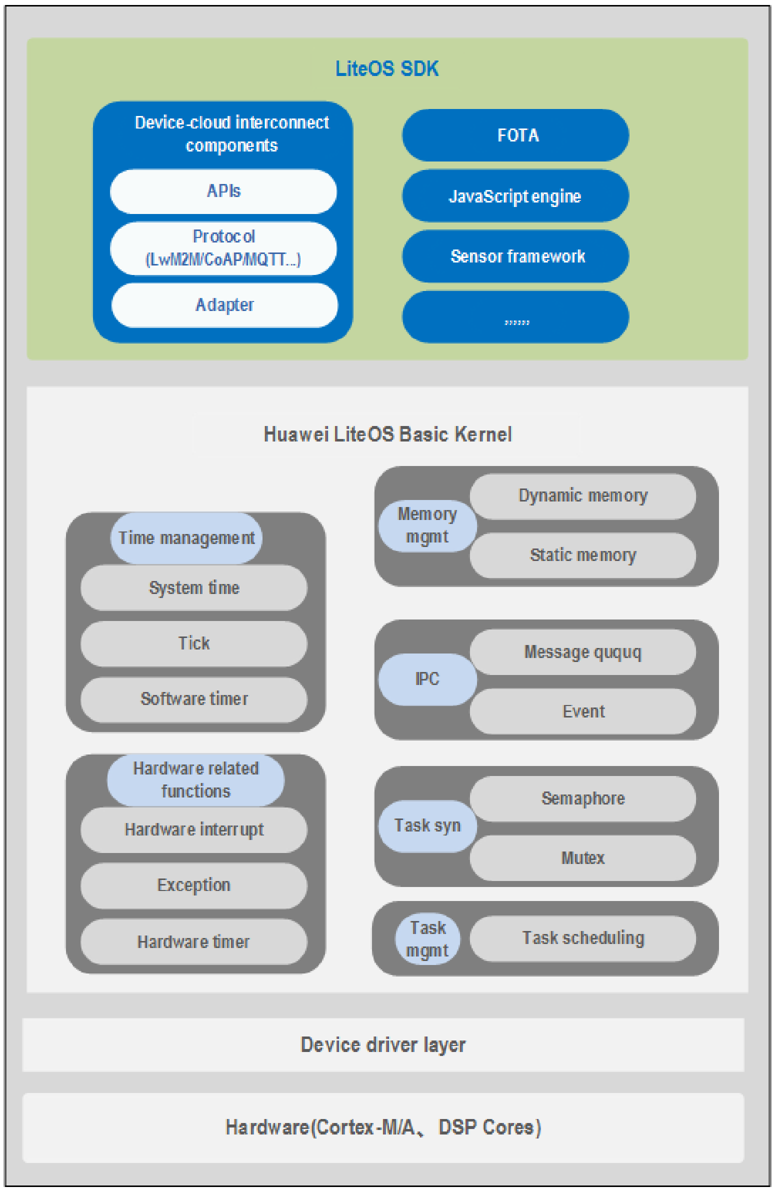
4.1.1 Overview

4.1.1.1 Background Introduction

LiteOS SDK consists of device-cloud interconnect components, FOTA, JavaScript engine, and sensor framework.

Device-cloud interconnect components are critical to connect devices with limited resources to OceanConnect in the Huawei IoT solution. Device-cloud interconnect components enable device-cloud synergy and integrate a full set of IoT interconnection protocol stacks, such as Lightweight M2M (LWM2M), Constrained Application Protocol (CoAP), mbed TLS, and lightweight IP (lwIP). Based on LWM2M, device-cloud interconnect components provide packaged open APIs for you to quickly and reliably connect applications to OceanConnect. In addition, they help you improve service development efficiency and quickly build products.

Figure 4-1 Huawei LiteOS architecture



4.1.1.2 System Plan

Device-cloud interconnect components provide the following two types of software architectures.

Figure 4-2 Architecture for single module or MCU

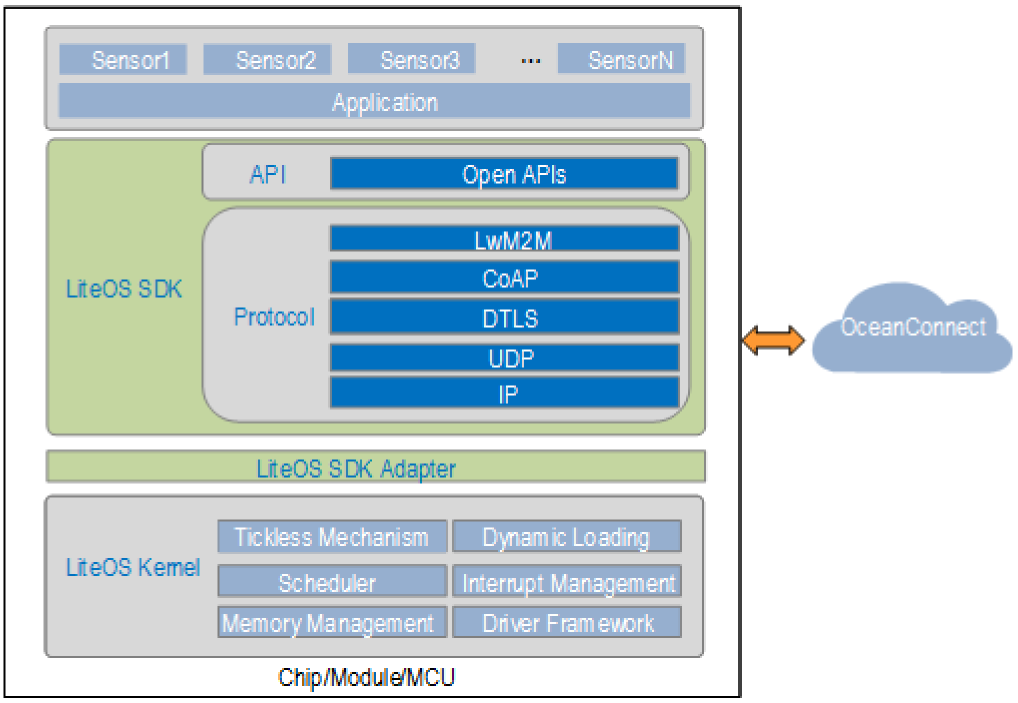
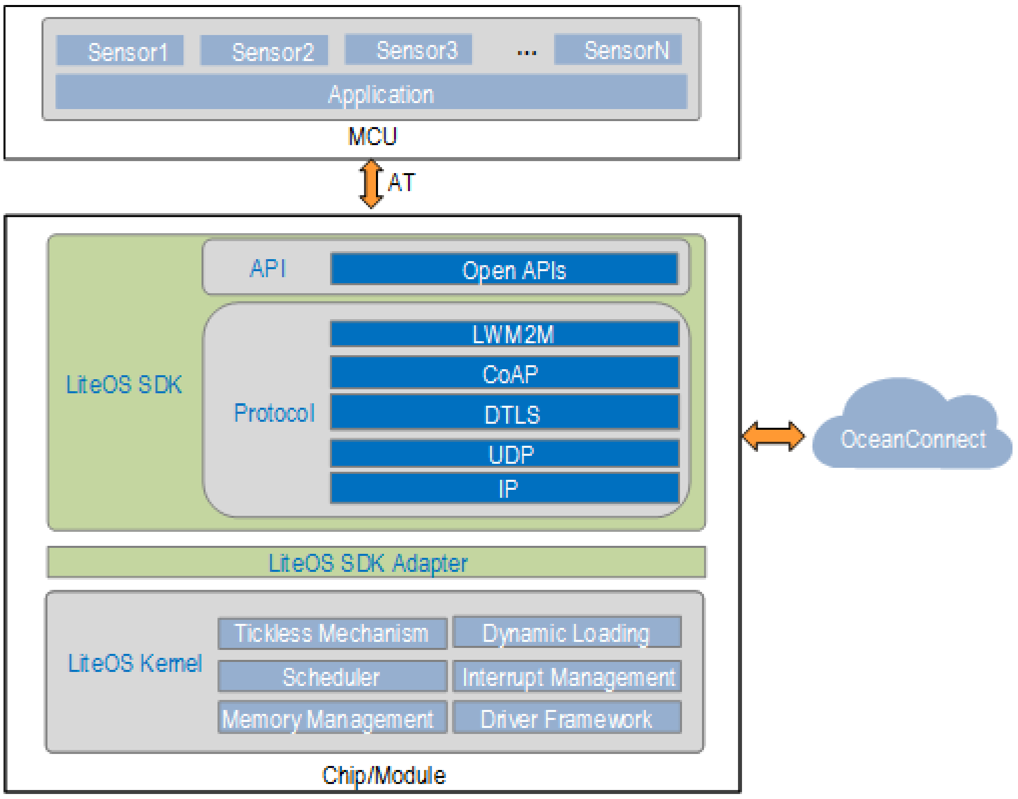


Figure 4-3 Architecture for external MCUs + chips/modules



Device-cloud interconnect components are divided into the following three layers:

- **Open API layer**: The device-cloud interconnect components provide open APIs for applications. Devices quickly connect OceanConnect, report service data, and process delivered commands by invoking these APIs. In the external MCUs + chips/modules scenario, device-cloud interconnect components also provides the AT instruction adaptation layer for parsing AT instructions.
- **Protocol layer**: Device-cloud interconnect components integrate protocols, such as LWM2M, CoAP, Datagram Transport Layer Security (DTLS), TLS, and UDP.
- **Driver and network adapter layer**: This layer facilitates device integration and porting. You can adapt to APIs related to the hardware random number, memory management, logs, data storage, and network sockets based on the API list of the adaptation layer provided by SDK and specific hardware platform.

LiteOS basic kernel provides RTOS features for devices.

4.1.1.3 Integration Strategies

4.1.1.3.1 Integrability

Device-cloud interconnect components can be easily integrated with various types of communications modules, such as NB-IoT, eMTC, Wi-Fi, GSM, and Ethernet hardware modules without considering the specific chip architecture and network hardware type.

4.1.1.3.2 Portability

The adapter layer of device-cloud interconnect components provides common hardware and network adapter APIs. Device or module vendors can complete the porting of device-cloud interconnect components after adapting their hardware to these APIs. The following table lists the to-be-ported APIs and related functions.

Table 4-1 APIs to which the to-be-ported device-cloud interconnect components need adapt

API Category	API	Description
Network socket API	atiny_net_connect	Creates a socket network connection.
	atiny_net_recv	Receives packets.
	atiny_net_send	Sends packets.
	atiny_net_recv_timeout	Receives packets in a blocking manner.
	atiny_net_close	Closes a socket network connection.
Hardware API	atiny_gettime_ms	Obtains the system time (ms).
	atiny_usleep	Delay function, measured in μs.
	atiny_random	Hardware random number function.

API Category	API	Description
	atiny_malloc	Applies for dynamic memory.
	atiny_free	Releases dynamic memory.
	atiny_snprintf	Formats character strings.
	atiny_printf	Outputs logs.
API for resource exclusion	atiny_mutex_create	Creates a mutual exclusion lock.
	atiny_mutex_destroy	Destroy a mutual exclusion lock.
	atiny_mutex_lock	Obtains a mutual exclusion lock.
	atiny_mutex_unlock	Releases a mutual exclusion lock.

NOTE

Device-cloud interconnect components can be ported in OS and non-OS modes. The OS mode is recommended.

Device-cloud interconnect components support firmware upgrade. The components need to adapt to the **atiny_storage_device_s** object.

```
atiny_storage_device_s *atiny_get_hal_storage_device(void);
struct atiny_storage_device_tag_s;
typedef struct atiny_storage_device_tag_s  atiny_storage_device_s;
struct atiny_storage_device_tag_s
{
//Device initialization
int (*init)( storage_device_s *this);
//Begin to write
int (*begin_software_download)( storage_device_s *this);
//Write software, and start from offset. buffer indicates the content, and len
indicates the length.
int (*write_software)( storage_device_s *this , uint32_t offset, const char
*buffer, uint32_t len);

//Download completed
int (*end_software_download)( storage_device_s *this);
//Activate software
int (*active_software)( storage_device_s *this);
//Activated results are obtained. 0 indicates successful. 1 indicates failed.
int (*get_active_result)( storage_device_s *this);
//Write update_info, and start from offset. buffer indicates the content, and len
indicates the length.
int (*write_update_info)( storage_device_s *this, long offset, const char
*buffer, uint32_t len);
//Read update_info, and start from offset. buffer indicates the content, and len
indicates the length.
int (*read_update_info)( storage_device_s *this, long offset, char *buffer,
uint32_t len);
};
```

4.1.1.3.3 Integration Restrictions

To integrate with device-cloud interconnect components, the following hardware specifications requirements must be met:

- Modules or chips are supported by physical network hardware and support the UDP protocol stack.
- Modules or chips provide sufficient Flash and RAM resources to integrate with protocol stacks for device-cloud interconnect components. The following table lists the recommended hardware specifications.

Table 4-2 Recommended hardware specifications

RAM	Flash
> 32 KB	> 128 KB

NOTE

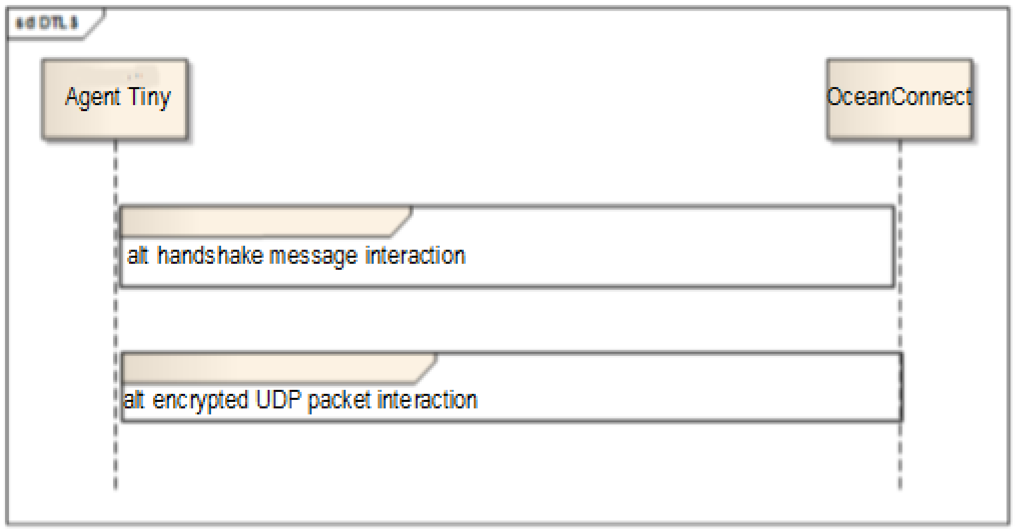
The recommended hardware specifications are determined based on resources (including open APIs, IoT protocol stacks, security protocols, SDK driver and network adapter layer) used by device-cloud interconnect components and resources (including chip drivers, sensor drivers, and basic service processes) minimally used by user service demos. The preceding specifications are for reference only. The specific hardware specifications need to be evaluated based on user service requirements.

4.1.1.4 Security

Device-cloud interconnect components support DTLS. Currently, the pre-shared key (PSK) mode is supported. Other modes will be supported.

After the components first complete the handshake process with OceanConnect, the subsequent application data will be encrypted, as shown in the following figure.

Figure 4-4 DTLS interaction process

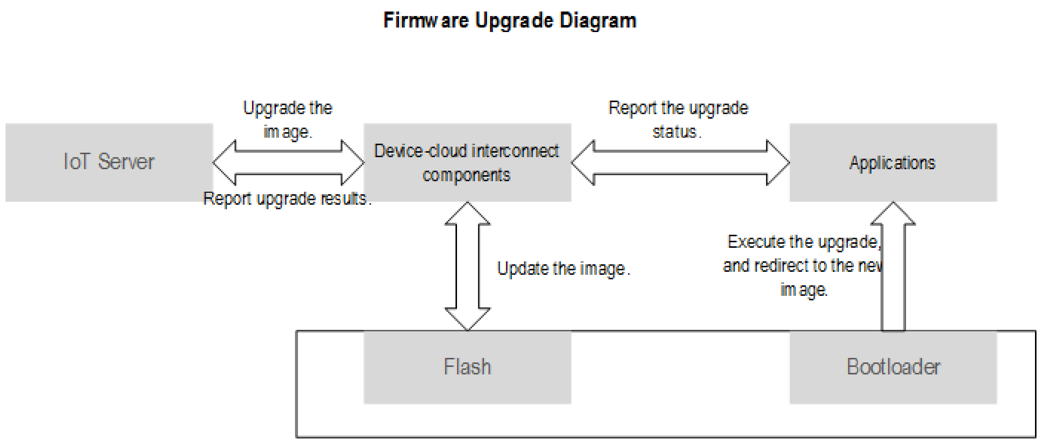

sd DTLS
Agent Tiny
OceanConnect
alt handshake message interaction
alt encrypted UDP packet interaction

4.1.1.5 Upgrade

Device-cloud interconnect components support the remote firmware upgrade of OceanConnect and feature resumable data transfer and firmware package integrity protection.

The following figure shows the firmware upgrade functions and process.

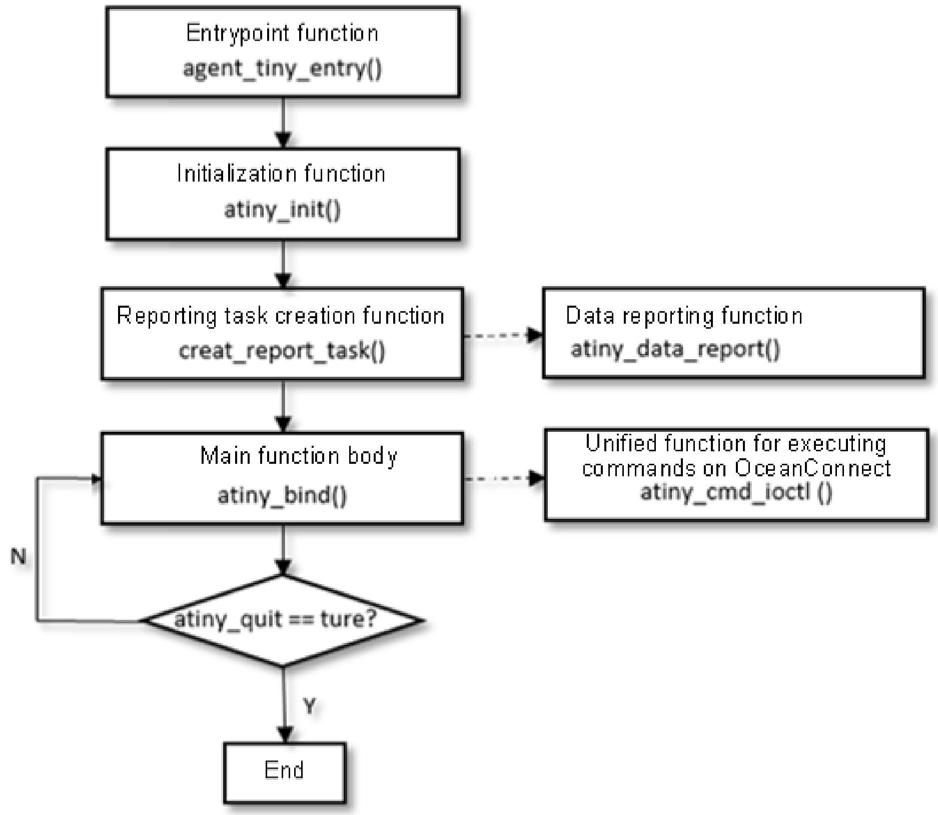
Figure 4-5 Firmware upgrade Diagram



4.1.2 Process for Connecting Devices to OceanConnect on the Device Side

When connecting a device to OceanConnect, ensure that the device has been registered and the device applications have been deployed on OceanConnect. After the device has been connected, OceanConnect can manage it. This section describes how to connect device-side devices to OceanConnect using device-cloud interconnect components. The following figure shows the general diagram of connecting device-side devices to OceanConnect.

Figure 4-6 General diagram of connecting device-side devices to OceanConnect



4.1.2.1 Preparations

The information to be obtained before development is as follows:

- Huawei LiteOS and LiteOS SDK source code. The general project architecture is as follows:

```
├── arch //Architecture-related files
|  ├── arm
|  └── msp430
├── build
|  └── Makefile
├── components //Various LiteOS components
|  ├── connectivity
|  ├── fs
|  ├── lib
|  ├── log
|  ├── net
|  ├── ota
|   └──── security
├──── demos //Sample programs
|   ├──── agenttiny_lwm2m //All sample programs listed in this chapter are from the
agent_tiny_demo.c file in this directory.
|   ├──── agenttiny_mqtt
|   ├──── dtls_server
|   ├──── fs
|   ├──── kernel
|   └──── nbiot_without_atiny
├──── doc //Documents
|   ├──── Huawei_LiteOS_Developer_Guide_en.md
|   ├──── Huawei_LiteOS_Developer_Guide_zh.md
|   ├──── Huawei_LiteOS_SDK_Developer_Guide.md
|   ├──── LiteOS_Code_Info.md
|   ├──── LiteOS_Commit_Message.md
|   ├──── LiteOS_Contribute_Guide_GitGUI.md
|   ├──── LiteOS_Supported_board_list.md
|   └──── meta
├──── include //Header files required by projects
|   ├──── at_device
|   ├──── at_frame
|   ├──── atiny_lwm2m
|   ├──── atiny_mqtt
|   ├──── fs
|   ├──── log
|   ├──── nb_iot
|   ├──── osdepends
|   ├──── ota
|   ├──── sal
|   └──── sota
├──── kernel //System kernels
|   ├──── base
|   ├──── extended
|   ├─── include
|   ├─── los_init.c
|   └─── Makefile
├─── LICENSE //Licenses
├─── osdepends //Dependencies
|   └─── liteos
├─── README.md
├─── targets //BSP projects
|   ├─── Cloud_STM32F429IGTx_FIRE
|   ├─── Mini_Project
|   ├─── NXP_LPC51U68
|   └─── STM32F103VET6_NB_GCC
└─── tests //Test cases
├─── cmockery
├─── test_agenttiny
├─── test_main.c
├─── test_sota
└─── test_suit
```

To obtain the source code, visit **https://github.com/LiteOS/LiteOS**.

- Integration development tools:
  - MDK 5.18 or later, which can be downloaded from http://www2.keil.com/mdk5
  - MDK packages

NOTE

The licenses for MDK tools can be obtained from http://www2.keil.com/mdk5.

4.1.2.2 Entrypoint Function for LiteOS SDK Device-Cloud Interconnect Components

To connect the LiteOS SDK device-cloud interconnect component Agent Tiny to OceanConnect, create an entrypoint function **agent_tiny_entry()**.

Function	Description

void agent_tiny_entry(void)	Entrypoint function for LiteOS SDK device-cloud interconnect components. This function can be used to initialize Agent Tiny, create report tasks, and call the main function body of Agent Tiny. Parameter list: N/A Return value: null

Based on the task mechanism provided by the LiteOS kernel, a developer can create a main task **main_task**, and call the entrypoint function **agent_tiny_entry()** in the main task to enable the Agent Tiny workflow.

```
UINT32 creat_main_task()
    {
        UINT32 uwRet = LOS_OK;
        TSK_INIT_PARAM_S task_init_param;
        task_init_param.usTaskPrio = 0;
        task_init_param.pcName = "main_task";
        task_init_param.pfnTaskEntry = (TSK_ENTRY_FUNC)main_task;
        task_init_param.uwStackSize = 0x1000;
        uwRet = LOS_TaskCreate(&g_TskHandle, &task_init_param);
        if(LOS_OK != uwRet)
        {
            return uwRet;
        }
        return uwRet;
    }
```

4.1.2.3 Initializing LiteOS SDK Device-Cloud Interconnect Components

Call the **atiny_init()** function in the entrypoint function to initialize Agent Tiny.

Function	Description
int atiny_init(atiny_param_t* atiny_params, void** phandle);	Function for initializing device-cloud interconnect components, which is implemented by device-cloud interconnect components and invoked by devices. The parameters involved are as follows: ● **atiny_params**. For details about the parameter, see the description of the **atiny_param_t** data structure. ● **phandle**, an output parameter, which represents the handle of the currently created device-cloud interconnect component. Return value: Integer variable, indicating that the initialization is successful or failed.

The input parameter **atiny_params** needs to be set based on specific services. Developers can set the parameter by the following code:

```
#ifdef CONFIG_FEATURE_FOTA
     hal_init_ota();   //To define the FOTA functions, perform FOTA-related
initialization.
 #endif

 #ifdef WITH_DTLS
     device_info->endpoint_name = g_endpoint_name_s;  //Encrypted device
verification code
 #else
     device_info->endpoint_name = g_endpoint_name;    //Unencrypted device
verification code
 #endif
 #ifdef CONFIG_FEATURE_FOTA
     device_info->manufacturer = "Lwm2mFota";    //Unencrypted device
verification code
     device_info->dev_type = "Lwm2mFota";        //Device type
 #else
     device_info->manufacturer = "Agent_Tiny";
 #endif
     atiny_params = &g_atiny_params;
     atiny_params->server_params.binding = "UQ";   //Binding mode
     atiny_params->server_params.life_time = 20;   //Life cycle
     atiny_params->server_params.storing_cnt = 0;  //Number of cached data packets

     atiny_params->server_params.bootstrap_mode = BOOTSTRAP_FACTORY;   //Boot mode
     atiny_params->server_params.hold_off_time = 10;    //Waiting latency

     //pay attention: index 0 for iot server, index 1 for bootstrap server.
     iot_security_param = &(atiny_params->security_params[0]);
     bs_security_param = &(atiny_params->security_params[1]);

     iot_security_param->server_ip = DEFAULT_SERVER_IPV4;  //Server address
     bs_security_param->server_ip = DEFAULT_SERVER_IPV4;

 #ifdef WITH_DTLS
     iot_security_param->server_port = "5684";   //Encrypted device port number
     bs_security_param->server_port = "5684";

     iot_security_param->psk_Id = g_endpoint_name_iots;         //Encrypted
device verification
     iot_security_param->psk = (char *)g_psk_iot_value;         //PSK password
     iot_security_param->psk_len = sizeof(g_psk_iot_value);     //PSK password
length

     bs_security_param->psk_Id = g_endpoint_name_bs;
     bs_security_param->psk = (char *)g_psk_bs_value;
     bs_security_param->psk_len = sizeof(g_psk_bs_value);
 #else
     iot_security_param->server_port = "5683";    //Unencrypted device port number
     bs_security_param->server_port = "5683";

     iot_security_param->psk_Id = NULL;    //No PSK-related parameter setting for
unencrypted devices
     iot_security_param->psk = NULL;
     iot_security_param->psk_len = 0;

     bs_security_param->psk_Id = NULL;
     bs_security_param->psk = NULL;
     bs_security_param->psk_len = 0;
 #endif
```

After setting the **atiny_params** parameter, initialize Agent Tiny based on the set parameter.

```
   if(ATINY_OK != atiny_init(atiny_params, &g_phandle))
    {
        return;
    }
```

After setting the **atiny_params** parameter, initialize Agent Tiny based on the set parameter.

4.1.2.4 Creating a Data Reporting Task

After initializing Agent Tiny, create a data reporting task function **app_data_report()** by calling the **creat_report_task()** function.

```
UINT32 creat_report_task()
 {
     UINT32 uwRet = LOS_OK;
     TSK_INIT_PARAM_S task_init_param;
     UINT32 TskHandle;
     task_init_param.usTaskPrio = 1;
     task_init_param.pcName = "app_data_report";
     task_init_param.pfnTaskEntry = (TSK_ENTRY_FUNC)app_data_report;
     task_init_param.uwStackSize = 0x400;
     uwRet = LOS_TaskCreate(&TskHandle, &task_init_param);
     if(LOS_OK != uwRet)
     {
         return uwRet;
     }
     return uwRet;
 }
```

In the **app_data_report()** function, assign a value to the reported data structure **data_report_t**, including the data buffer address **buf**, callback function **callback** called after the ACK response is received from a platform, data **cookie**, data length **len**, and data reporting type **type** (set to **APP_DATA** by default).

```
uint8_t buf[5] = {0, 1, 6, 5, 9};
 data_report_t report_data;
 int ret = 0;
 int cnt = 0;
 report_data.buf = buf;
 report_data.callback = ack_callback;
 report_data.cookie = 0;
 report_data.len = sizeof(buf);
 report_data.type = APP_DATA;
```

After a value is assigned to the **report_data** parameter, data can be reported by calling the **atiny_data_report()** function.

Function	Description
int atiny_data_report(void* phandle, data_report_t* report_data)	Function for reporting data of device-cloud interconnect components, which is implemented by device-cloud interconnect components and invoked by devices. This function is used to report device application data. The function is blocked and cannot be used when being interrupted. The parameters involved are as follows: Parameter list: **phandle** is the Agent Tiny handle obtained by calling the initialization function **atiny_init()**. **report_data** is the reported data structure. Return value: Integer variable, indicating that the data reporting is successful or failed.

The implementation method of a report task in the sample code is as follows:

```
while(1)
 {
     report_data.cookie = cnt;
     cnt++;
     ret = atiny_data_report(g_phandle, &report_data);   //Data reporting
function
        ATINY_LOG(LOG_DEBUG, "data report ret: %d\n", ret);
        (void)LOS_TaskDelay(250 * 8);
    }
```

4.1.2.5 Command Processing Function for LiteOS SDK Device-Cloud Interconnect Components

All commands delivered by OceanConnect are executed by calling the **atiny_cmd_ioctl()** function.

Function	Description
int atiny_cmd_ioctl (atiny_cmd_e cmd, char* arg, int len);	Implemented by developers to declare and invoke device-cloud interconnect components. This API is a unified portal for LWM2M standard objects to deliver commands to devices. The parameters involved are as follows: ● **cmd**, a specific command word, such as commands for delivering service data and resetting and upgrade. ● **arg**, a specific command parameter; **len**, the parameter length. Return value: null

The **atiny_cmd_ioctl** API is a universal extensible API defined by device-cloud interconnect components. The command word of this API is defined by referring to the enumerated type **atiny_cmd_e**. Users can implement or extend this API based on respective requirements. The following table lists common APIs. Each API corresponds to an enumerated value of the **atiny_cmd_e** API.

Callback Function	Description
int atiny_get_manufacturer(char* manufacturer,int len)	Obtains the vendor name. The memory specified by the **manufacturer** parameter is allocated by device-cloud interconnect components. A user can specify the parameter. The parameter length cannot exceed the value of **len**.
int atiny_get_dev_type(char * dev_type,int len)	Obtains the device type. The memory specified by the **dev_type** parameter is allocated by device-cloud interconnect components. A user can specify the parameter. The parameter length cannot exceed the value of **len**.
int atiny_get_model_number((char * model_numer, int len)	Obtains the device model number. The memory specified by the **model_number** parameter is allocated by device-cloud interconnect components. A user can specify the parameter. The parameter length cannot exceed the value of **len**.
int atiny_get_serial_number(char* num,int len)	Obtains the device SN. The memory specified by the **number** parameter is allocated by device-cloud interconnect components. A user can specify the parameter. The parameter length cannot exceed the value of **len**.

Callback Function	Description
int atiny_get_dev_err(int* arg,int len)	Obtains the device status, such as used-up memory, low battery, and low signal strength. The **arg** parameter is allocated by device-cloud interconnect components. A user can specify the parameter. The parameter length cannot exceed the value of **len**.
int atiny_do_dev_reboot(void)	Resets devices.
int atiny_do_factory_reset(void)	Resets vendors.
int atiny_get_baterry_level(int* voltage)	Obtains remaining battery level.
int atiny_get_memory_free(int* size)	Obtains available memory size.
int atiny_get_total_memory(int* size)	Obtains total memory size.
int atiny_get_signal_strength(int* singal_strength)	Obtains signal strength.
int atiny_get_cell_id(long * cell_id)	Obtains the cell ID.
int atiny_get_link_quality(int* quality)	Obtains the channel quality.
int atiny_write_app_write(void* user_data, int len)	Delivers service data.
int atiny_update_psk(char * psk_id, int len)	Updates PSKs.

A developer needs to make a command response by calling the **atiny_write_app_write()** function based on site services.

```
    int atiny_write_app_write(void* user_data, int len)
     {
         (void)atiny_printf("write num19 object success\r\n");
```

```
        return ATINY_OK;
    }
```

4.1.2.6 Main Function Body for LiteOS SDK Device-Cloud Interconnect Components

After creating the data reporting task and implementing the command processing function, call the **atiny_bind()** function.

Function	Description
int atiny_bind(atiny_device_info_t* device_info, void* phandle)	Main function body of a device-cloud interconnect component, which is implemented by device-cloud interconnect components and invoked by devices. However, no value is returned after the function is successfully called. This function is the main loop body of a device-cloud interconnect component, which implements LWM2M processing, state machine registration, queue retransmission, and subscription reporting. Parameter list: **device_info** is the device parameter structure. **phandle** is the Agent Tiny handle obtained by calling the initialization function **atiny_init()**. Return value: Integer variable, indicating the execution status of the main function body for LiteOS SDK device-cloud interconnect components. This value can be returned only when the execution failed or the deinitialization function **atiny_deinit()** for LiteOS SDK device-cloud interconnect components is called.

The **atiny_bind()** function can be used to create and register the LwM2M client based on the LwM2M protocol, send the data reported in the data reporting task creation function **app_data_report()** to OceanConnect through communication modules, receive and parse commands delivered by OceanConnect, and submit the parsed commands to the command processing function **atiny_cmd_ioctl()** for unified processing. Similar to the **atiny_init()** function, the **atiny_bind()** function does not need to be modified by developers.

NOTE

For details about the LWM2M protocol, see the appendix.

LiteOS SDK device-cloud interconnect components continuously report data and process commands through the main function body. When calling the deinitialization function **atiny_deinit()** for LiteOS SDK device-cloud interconnect components, exit the main function body.

Function	Description

void atiny_deinit(void* phandle);	Function for deinitializing device-cloud interconnect components, which is implemented by device-cloud interconnect components and invoked by devices. This function is blocked. It cannot stop being invoking until the main task of Agent Tiny quits and resources are completely released. Parameter list: **phandle** is the LiteOS SDK device-cloud interconnect component handle obtained by calling the **atiny_init()** function. Return value: null

4.1.2.7 Data Structure

- Enumerated type of commands delivered by OceanConnect

```
 typedef enum
  {
      ATINY_GET_MANUFACTURER,         /*Obtain the manufacturer name.*/
      ATINY_GET_MODEL_NUMBER,         /*Obtain device models defined and used by
the manufacturer.*/
      ATINY_GET_SERIAL_NUMBER,        /*Obtain the device SN.*/
      ATINY_GET_FIRMWARE_VER,         /*Obtain the firmware version number.*/
      ATINY_DO_DEV_REBOOT,            /*Deliver device resetting commands.*/
      ATINY_DO_FACTORY_RESET,         /*Restore factory resetting.*/
      ATINY_GET_POWER_SOURCE,         /*Obtain power supplies.*/
      ATINY_GET_SOURCE_VOLTAGE,       /*Obtain device voltage.*/
      ATINY_GET_POWER_CURRENT,        /*Obtain device current.*/
      ATINY_GET_BATERRY_LEVEL,        /*Obtain the battery level.*/
      ATINY_GET_MEMORY_FREE,          /*Obtain idle memory.*/
      ATINY_GET_DEV_ERR,              /*Obtain the device status, such as used-up
memory and low battery level.*/
      ATINY_DO_RESET_DEV_ERR,         /*Obtain the device resetting status.*/
      ATINY_GET_CURRENT_TIME,         /*Obtain the current time.*/
      ATINY_SET_CURRENT_TIME,         /*Set the current time.*/
      ATINY_GET_UTC_OFFSET,           /*Obtain the UTC difference.*/
      ATINY_SET_UTC_OFFSET,           /*Set the UTC difference.*/
      ATINY_GET_TIMEZONE,             /*Obtain the time zone.*/
      ATINY_SET_TIMEZONE,             /*Set the time zone.*/
      ATINY_GET_BINDING_MODES,        /*Obtain the binding mode.*/
      ATINY_GET_FIRMWARE_STATE,       /*Obtain the firmware upgrade status.*/
      ATINY_GET_NETWORK_BEARER,       /*Obtain the network bearer type, such as
GSM and WCDMA. */
      ATINY_GET_SIGNAL_STRENGTH,      /*Obtain the network signal strength.*/
      ATINY_GET_CELL_ID,              /*Obtain the network cell ID.*/
      ATINY_GET_LINK_QUALITY,         /*Obtain network link quality.*/
      ATINY_GET_LINK_UTILIZATION,     /*Obtain network link usage.*/
      ATINY_WRITE_APP_DATA,           /*Write command words delivering service
data.*/
      ATINY_UPDATE_PSK,               /*Update PSK command words.*/
      ATINY_GET_LATITUDE,             /*Obtain device latitude.*/
      ATINY_GET_LONGITUDE,            /*Obtain device longitude.*/
      ATINY_GET_ALTITUDE,             /*Obtain device height.*/
      ATINY_GET_SPEED,                /*Obtain device running speed.*/
      ATINY_GET_TIMESTAMP,            /*Obtain timestamp.*/
  } atiny_cmd_e;
```

- Enumerated type of key events

This enumerated type is used to notify users of the statuses of LiteOS SDK device-cloud interconnect components.

```
typedef enum
 {
     ATINY_REG_OK,               /*Device registration successful*/
     ATINY_REG_FAIL,             /*Device registration failed*/
     ATINY_DATA_SUBSCRIBLE,      /*Starting data subscription. Devices allow to
report data */
     ATINY_DATA_UNSUBSCRIBLE,   /*Canceling data subscription. Devices stop
reporting data*/
     ATINY_FOTA_STATE            /*Firmware upgrade status*/
 } atiny_event_e;
```

- LwM2M parameter structure

```
typedef struct
 {
     char* binding;                                /*U or UQ is currently
supported.*/
     int   life_time;                              /*LwM2M protocol life cycle,
which is set to 50000 by default.*/
     unsigned int  storing_cnt;                    /*Number of LwM2M cache data
packets*/
 } atiny_server_param_t;
```

- Security and server parameter structure

```
typedef struct
 {
     bool  is_bootstrap;      /*Whether the bootstrap server is used.*/
     char* server_ip;         /*Server IP address, which can be represented by
character strings and supports IPv4 and IPv6.*/
     char* server_port;       /*Server port number.*/
     char* psk_Id;            /*PSK ID.*/
     char* psk;               /*PSK*/
     unsigned short psk_len;  /*PSK length*/
 } atiny_security_param_t;
```

- Enumerated type of reported data

Type of data reported by users, which can be expanded based on users' applications.

```
typedef enum
 {
     FIRMWARE_UPDATE_STATE = 0;  /*LWM2M protocol life cycle, which is set to
50000 by default.*/
     APP_DATA                      /*User data*/
 } atiny_report_type_e;
```

- Server parameter structure

```
typedef struct
 {
     atiny_server_param_t   server_params;
     atiny_security_param_t security_params[2];  /*One IoT server and one
bootstrap server are supported.*/
 } atiny_param_t;
```

- Device parameter structure

```
typedef struct
 {
     char* endpoint_name;    /*Device ID generated for northbound application*/
     char* manufacturer;     /*Manufacturer name generated for northbound
application*/
     char* dev_type;         /*Device type generated for northbound application*/
 } atiny_device_info_t;
```

- Reported data structure

The following enumerated values indicate user data types. For example, data is sent successfully; data has been sent but is not acknowledged. The specific information is as follows:

```
typedef enum
 {
     NOT_SENT = 0,         /*To-be-reported data has not been sent.*/
     SENT_WAIT_RESPONSE,  /*To-be-reported data has been sent and is waiting for
response.*/
     SENT_FAIL,           /*To-be-reported data sending failed.*/
     SENT_TIME_OUT,       /*To-be-reported data has been sent and waiting for
response times out.*/
     SENT_SUCCESS,        /*To-be-reported data sending successful.*/
     SENT_GET_RST,        /*To-be-reported data has been sent but the receiver
sends an RST packet.*/
     SEND_PENDING,        /*To-be-reported data is waiting for sending.*/
 } data_send_status_e;
```

//Users can use the following data structure to report data:

```
  typedef struct _data_report_t
 {
     atiny_report_type_e type;    /*Reported data type, such as service data and
remaining battery level.*/
     int cookie;                  /*Data cookie, which is used to distinguish
data during ACK callback.*/
     int len;                     /*Data length, which must be not greater than
MAX_REPORT_DATA_LEN.*/
     uint8_t* buf;                /*First address of the data buffer.*/
     atiny_ack_callback callback; /*ACK callback, whose value is
data_send_status_e.*/
 } data_report_t;
```

4.1.3 Appendix 1 LWM2M

4.1.3.1 Definition

LWM2M is a lightweight, standard, and general-purpose IoT device management protocol developed by the Open Mobile Alliance (OMA). It can be used to quickly deploy IoT services in client or server mode.

In addition, LWM2M provides a set of standards for the management and application of IoT devices. It supports small and portable security communications APIs and efficient data models to implement M2M device management and service support.

4.1.3.2 Features

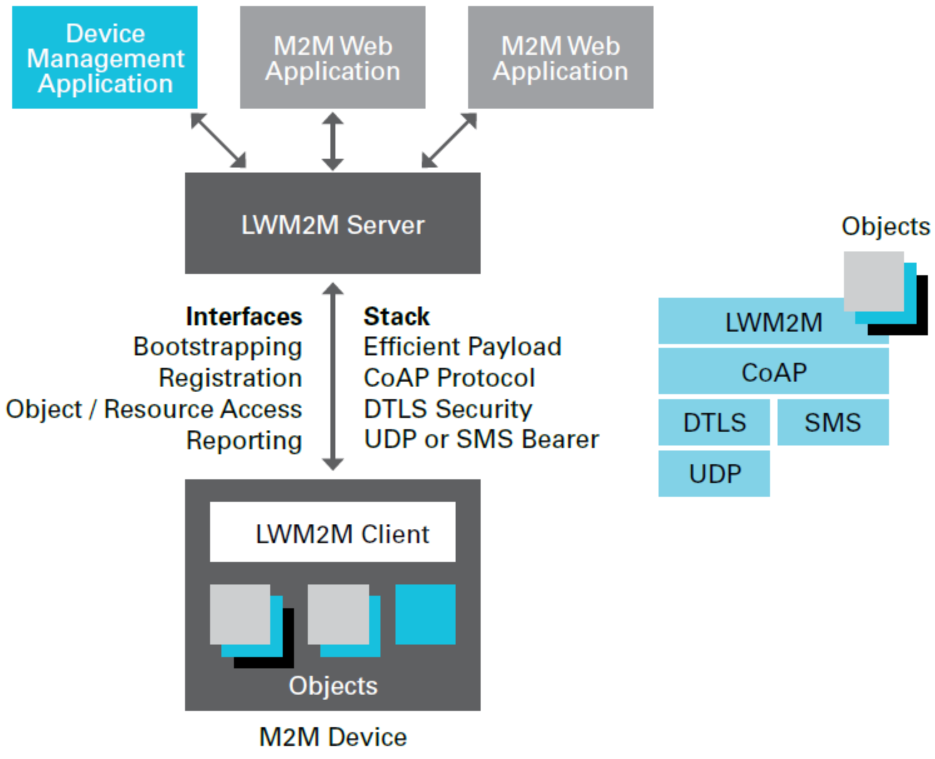
LWM2M supports the following features:

- Simple objects based on resource models
- Resource operations including creation, retrieval, update, deletion, and attribute configuration
  - Resource observation or notification
- Data formats including TLV, JSON, plain text, and opaque
- Transport layer protocols including UDP and SMS
- DTLS
- NAT or firewall solution — queue mode
- Multiple LWM2M Servers
- Basic M2M functions including LWM2M Server, Access Control, Devices, Connectivity Monitoring, Firmware, Location, and Connectivity Statistics

4.1.3.3 System Architecture

The following figure shows the system architecture of LWM2M.

Figure 4-7 System architecture of LWM2M



4.1.3.4 Object Defined by LWM2M

Object Concept

An object is a collection of resources that are logically used for specific purposes. For example, firmware upgrade. The object includes all resources used for firmware upgrade, such as firmware packages, firmware URLs, upgrade execution, and upgrade results.

Before using the functions of an object, instantiate the object. An object can have multiple instances, which are numbered from 0 in ascending order.

LWM2M has defined fixed IDs for the standard objects defined by the OMA. For example, the ID of the firmware upgrade object is 5. The object includes eight types of resources, which are numbered from 0 to 7. The ID of the firmware package name is 6. Therefore, URI 5/0/6 represents the firmware package name of instance 0 of the firmware upgrade object.

Object Format

Name	Object ID	Instance	Mandatory	Object URN
Object Name	16-bit Unsigned Integer	Multiple/Single	Mandatory/Optional	urn:oma:LwM2M:{oma,ext,x}:{Object ID}

Standard Object Defined by OMA

The OMA LWM2M specifications define the following seven standard objects.

Object	Object ID	description
LwM2M Security	0	Includes the URI and payload security mode of an LWM2M bootstrap server and information about partial algorithms or keys and short server IDs.
LwM2M Server	1	Includes the short ID of a server, registration life cycle, minimum or maximum period of observation, and binding models.
Access Control	2	Includes the access control permission of each object.
Device	3	Includes the device manufacturer, model, serial number, power, and memory.
Connectivity Monitoring	4	Includes the network standard, link quality, and IP address.
Firmware	5	Includes the firmware package and its URI, status, and upgrade results.
Location	6	Includes the latitude, longitude, altitude, and time stamp.
Connectivity Statistics	7	Includes the data volume sent and received during data collection and package size.

Device-cloud interconnect components match OceanConnect capabilities and support the following LWM2M APPDATA with the object ID of 19.

Object	Object ID	Description
LwM2M APPDATA	19	Includes application service data on LWM2M servers, such as water meter data.

NOTE

For details about other common objects defined by the OMA, see **http://www.openmobilealliance.org/wp/OMNA/LwM2M/LwM2MRegistry.html**.

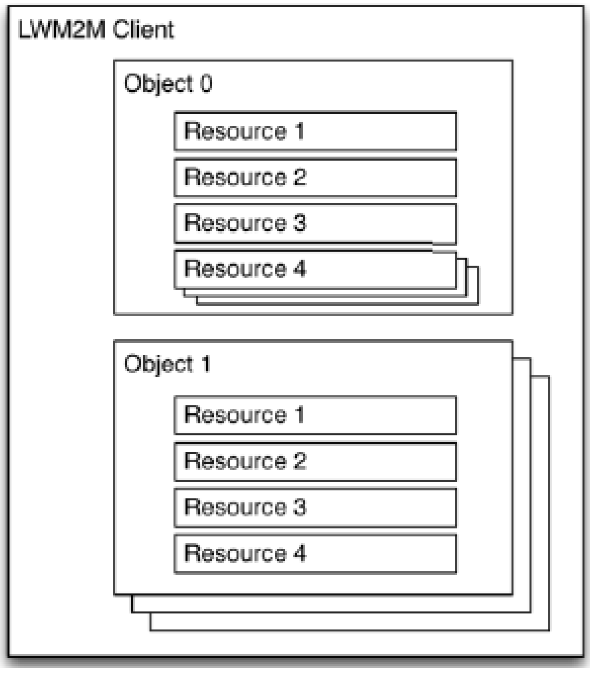
4.1.3.5 Resource Defined by LWM2M

Resource Model

LWM2M defines a resource model. In this resource model, all information can be abstracted and accessed as resources. An object includes resources. An LWM2M Client can have a large amount of resources. Like an object, a resource can have multiple instances.

The following figure shows the relationship among the LWM2M Client, objects, and resources.

Figure 4-8 Relationship among the LWM2M Client, objects, and resources



Resource Format

ID	0
Name	Resource Name
Operation	R (Read), W (Write), E (Execute)
Instance	Multiple/Single
Mandatory	Mandatory/Optional

Type	String, Integer, Float, Boolean, Opaque, Time, Objlnk none
Range or Enumeration	If any
Unit	If any
Description	Description

4.1.3.6 API Defined by LWM2M

Overview

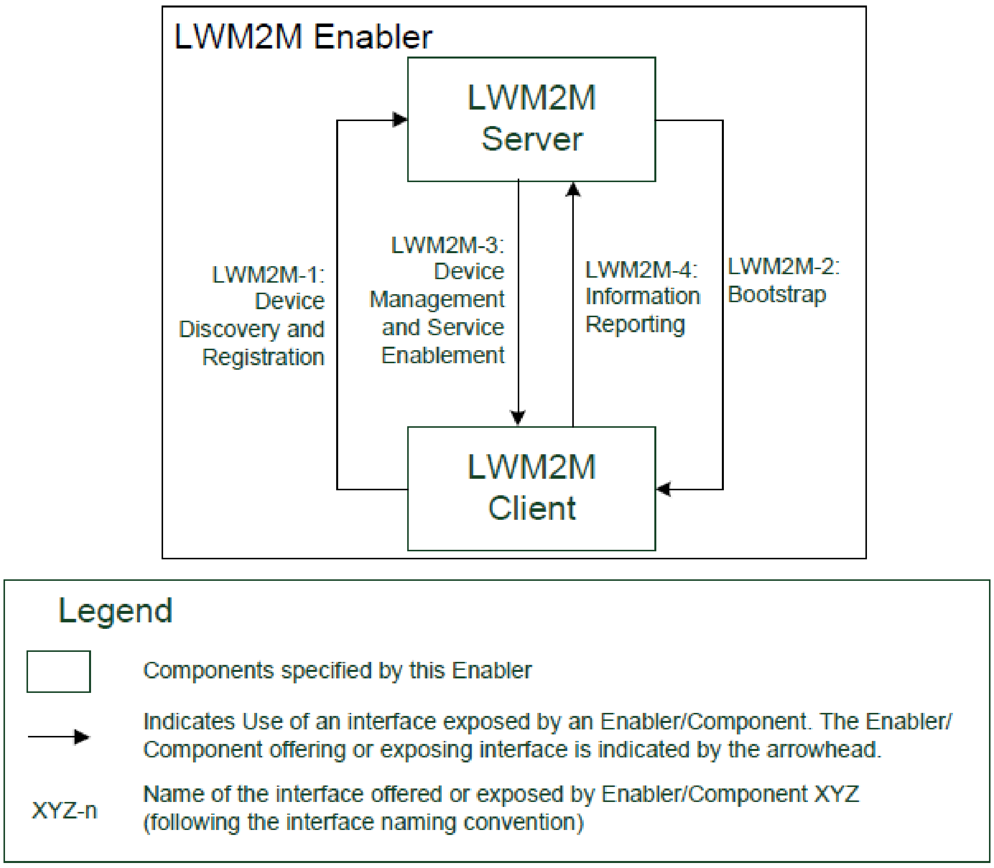
The LWM2M Enabler consists of two components: LWM2M Server and LWM2M Client. LWM2M designs the following four types of APIs for the interaction between the two components:

- API for device discovery and registration
- Bootstrap API
- API for device management and service enablement
- Information reporting API

API Model

The following figure shows an API model defined by LWM2M.

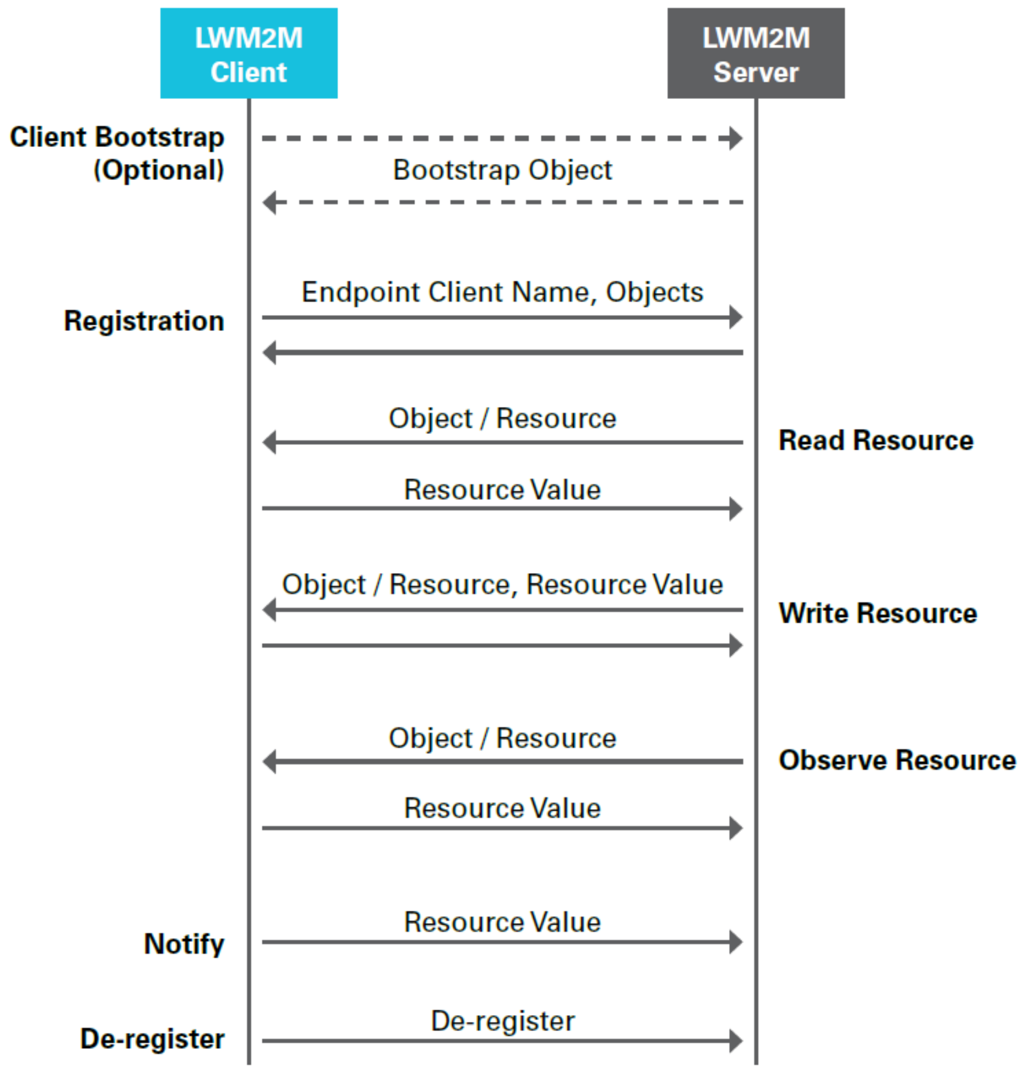
Figure 4-9 API model defined by LWM2M


LWM2M Enabler
LWM2M Server
LWM2M Client
LWM2M-1: Device Discovery and Registration
LWM2M-3: Device Management and Service Enablement
LWM2M-4: Information Reporting
LWM2M-2: Bootstrap
Legend
Components specified by this Enabler
Indicates Use of an interface exposed by an Enabler/Component. The Enabler/ Component offering or exposing interface is indicated by the arrowhead.
XYZ-n
Name of the interface offered or exposed by Enabler/Component XYZ (following the interface naming convention)

Message Interaction Process

The following figure shows the message interaction process defined by LWM2M.

Figure 4-10 Message interaction process defined by LWM2M



API for Device Management and Service Enablement

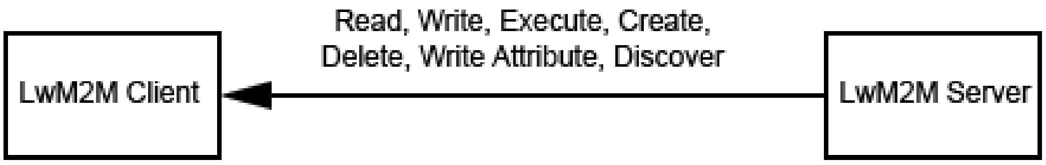
Each type of LWM2M APIs represents a type of functions. The API for device management and service implementation is one of the four types of APIs defined by LWM2M.

The functions of the four types of APIs are implemented by the following two operations:

- Upstream operation: LWM2M Client －> LWM2M Server
- Downstream operation: LWM2M Server －> LWM2M Client

LWM2M Server accesses object instances and resources of the LWM2M Client through the API for device management and service enablement. This API implements seven operations including create, read, write, delete, execute, write attributes, and discover.

Figure 4-11 Operations implemented by the API for device management and service enablement


Read, Write, Execute, Create,
Delete, Write Attribute, Discover
LwM2M Client
LwM2M Server

API	Operation	Direction
Device management and service enablement	Create, read, write, delete, execute, write attributes, and discover	Downstream

The following figure shows the interaction process implemented by the API for device management and service enablement.

Figure 4-12 Interaction process implemented by the API for device management and service enablement

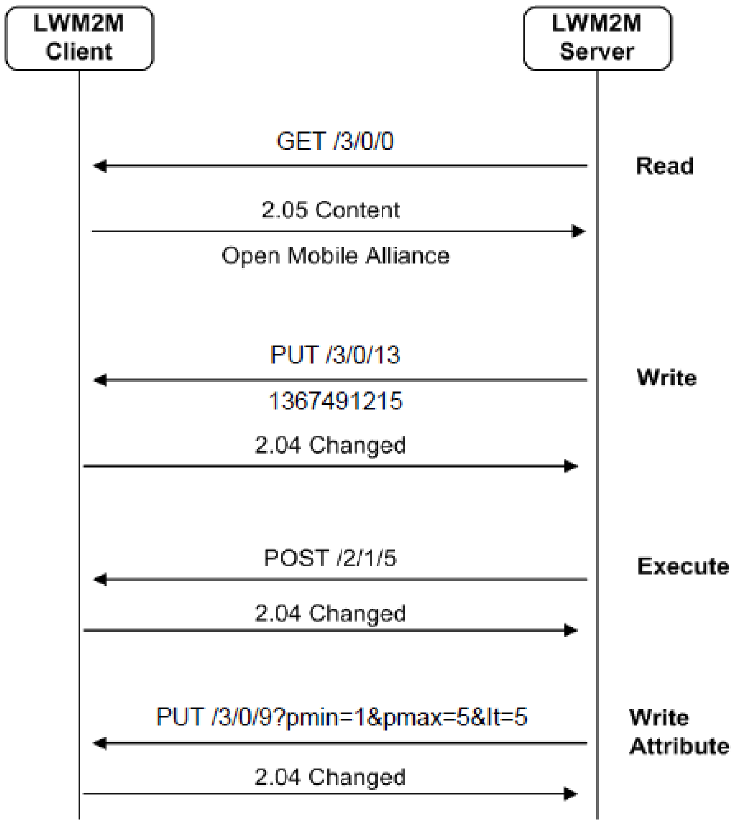
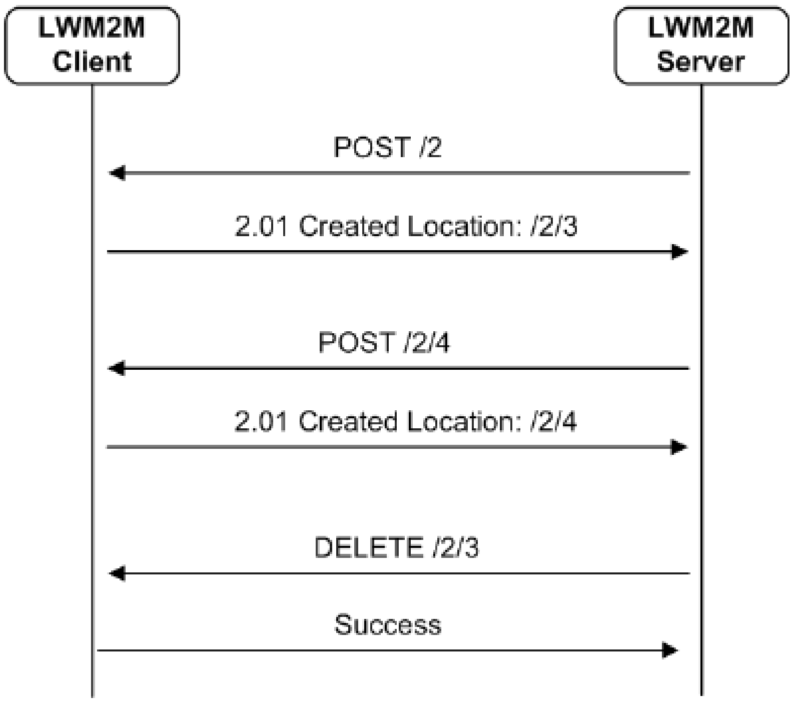


Figure 4-13 Creating and deleting an object



4.1.3.7 Firmware Upgrade

The firmware upgrade object makes it possible for users to manage the firmware upgrade. The firmware upgrade objects include installing the firmware package, updating the firmware, and other actions. After the firmware is successfully upgraded, the corresponding device must be restarted to make the new firmware take effect.

Before the device is restarted, values related to the upgrade results must be saved.

After the device is restarted, if the **Packet** resource contains a valid but uninstalled firmware package, the **State** resource must be in the downloaded state. Otherwise, it must be in the idle state.

Object Definition

Name	Object ID	Instance	Mandatory	Object URN
Firmware Update	5	Single	Optional	rn:oma:LwM2M:oma:5

Resource Definition

ID	Name	Operation	Instance	Mandatory	Type	Range or Enumeration	Description
0	Package	W	Single	Mandatory	Opaque		Firmware package.

ID	Name	Operation	Instance	Mandatory	Type	Range or Enumeration	Description
1	Package URI	W	Single	Mandatory	String	0-255 bytes	URI for downloading the firmware package.
2	Update	E	Single	Mandatory	none	no argument	Updating the firmware. The resource is executable only when the **State** resource is in the downloaded state.
3	State	R	Single	Mandatory	Integer	0-3	Firmware upgrade status. The value is set by the LWM2M Client. 0: Four statuses of the firmware are as follows: **Idle**, **Downloading**, **Downloaded**, and **Updating**. If the **Resource Update** command is executed, the status changes from **Downloaded** to **Updating**. If the upgrade is successful, the status changes to **Idle**. If the upgrade fails, the status changes to **Downloaded**.

ID	Name	Operation	Instance	Mandatory	Type	Range or Enumeration	Description
4	Update Supported Objects	RW	Single	Optional	Boolean		The default value is **false**. If the value is set to **true**, the LWM2M Client must notify the LWM2M Server of the **Object** parameter value change by sending the upgrade message or registration message after the firmware is successfully upgraded. If the upgrade fails, the **Object** parameter value change is reported by sending the upgrade message in the next phase.

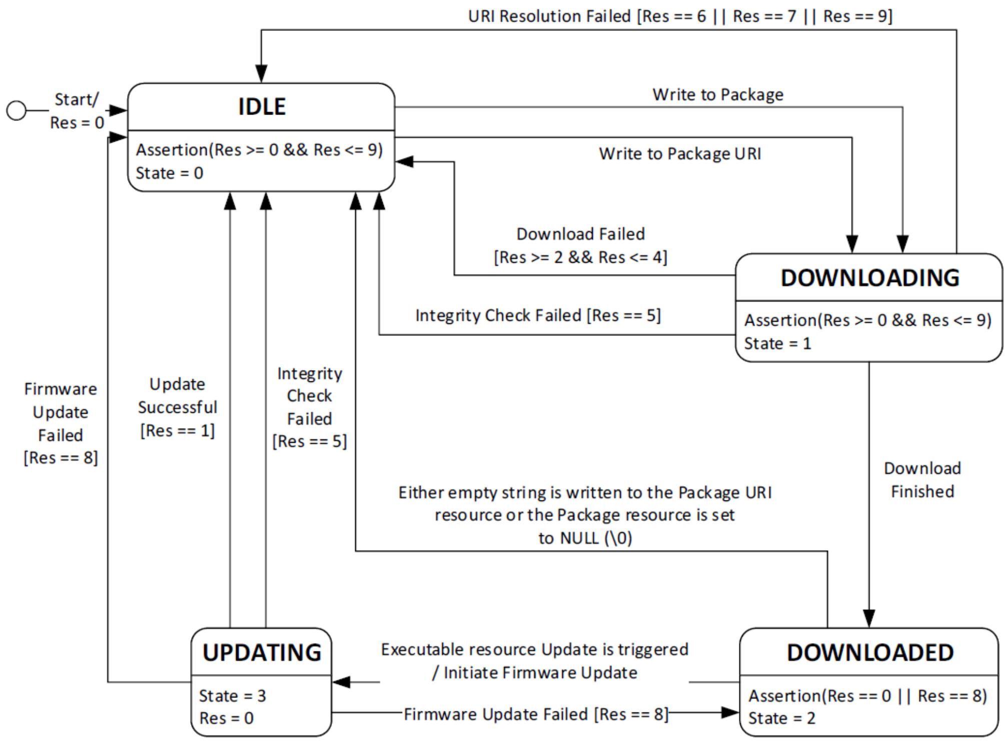
ID	Name	Operation	Instance	Mandatory	Type	Range or Enumeration	Description
5	Update Result	R	Single	Mandatory	Integer	0-8	The results of downloading or upgrading the firmware are as follows:0: Initial value. When upgrade or downloading starts, the resource value must be set to **0**. 1: The firmware is successfully upgraded; 2: The space for storing the new firmware package is insufficient; 3: The memory is insufficient in the downloading process; 4: The connection breaks in the downloading process; 5: Failed to check the integrity of the newly downloaded package; 6: Unsupported package types; 7: Invalid URI; 8: The firmware upgrade fails, and this resource can be reported by executing the **Observe** command.
6	PkgName	R	Single	Optional	String	0-255 bytes	Name of the firmware package.

ID	Name	Operation	Instance	Mandatory	Type	Range or Enumeration	Description
7	PkgVersion	R	Single	Optional	String	0-255 bytes	Version of the firmware package.

Status Mechanism

The following figure shows the firmware upgrade status mechanism.

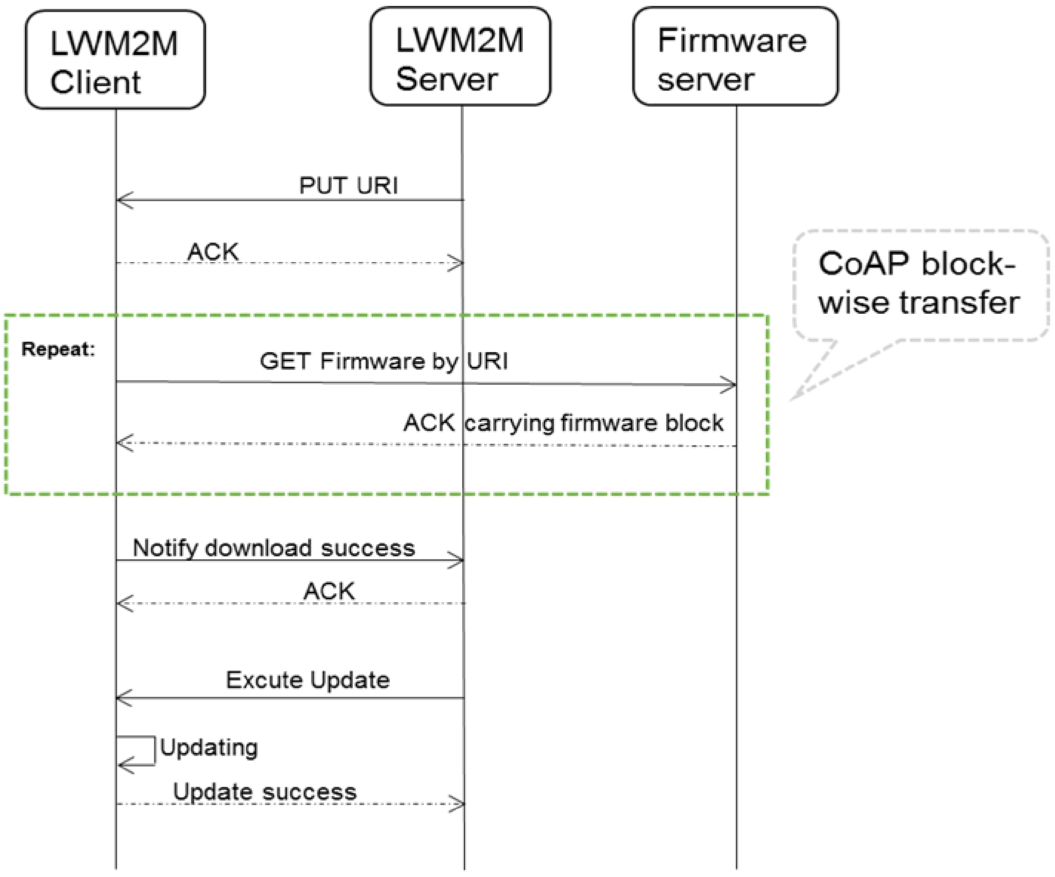
Figure 4-14 Firmware upgrade status mechanism


URI Resolution Failed [Res == 6 || Res == 7 || Res == 9]
Start/
Res = 0
IDLE
Assertion(Res >= 0 && Res <= 9)
State = 0
Write to Package
Write to Package URI
Download Failed
[Res >= 2 && Res <= 4]
Integrity Check Failed [Res == 5]
DOWNLOADING
Assertion(Res >= 0 && Res <= 9)
State = 1
Firmware Update Failed [Res == 8]
Update Successful [Res == 1]
Integrity Check Failed [Res == 5]
Either empty string is written to the Package URI resource or the Package resource is set to NULL (\0)
Download Finished
UPDATING
State = 3
Res = 0
Executable resource Update is triggered / Initiate Firmware Update
Firmware Update Failed [Res == 8]
DOWNLOADED
Assertion(Res == 0 || Res == 8)
State = 2

Flowchart

The following figure shows the firmware upgrade flowchart.

Figure 4-15 Firmware upgrade flowchart



5 SDK Usage Guide on the Application Side

5.1 Huawei IoT Platform Java SDK Usage Guide

5.1.1 Before You Start

- This document describes how to use the Java SDK to connect to the IoT platform, such as certificate configuration and callback.
- The northbound Java SDK Demo is used as an example. Each class (except the tool class) contains a main method, which can be run independently to demonstrate how to call SDK APIs.

5.1.2 Requirements for the Development Environment

Requirements for the development environment

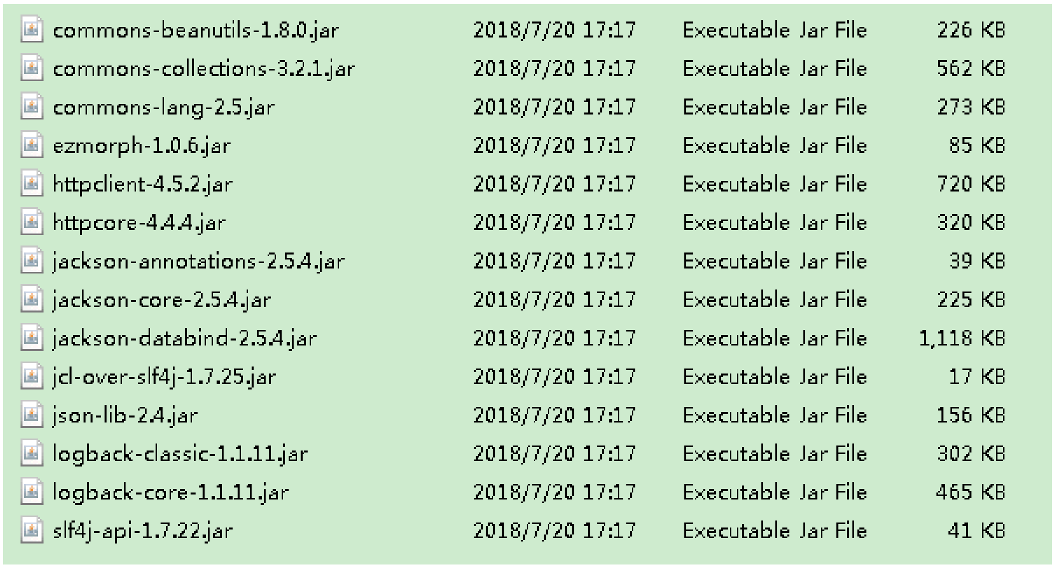
Development Platform	Development Environment	Mapping Requirements	Recommended OS
IoT	1. J2EE for Java Developers 2. **Maven** plug-in: m2e-Maven integration for **Eclipse** (includes incubating components)	JDK 1.8 or later	Windows 7

The SDK packages are pure Java JAR packages. They do not have any special limitations as long as the JDK version is 1.8 or later.

5.1.3 Downloading Related Development Resources

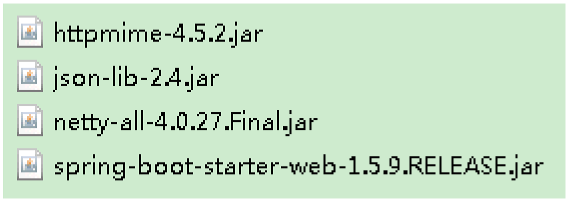
Obtain the northbound Java SDK Demo and northbound Java SDK from **Resources**.

- The Java SDK is stored in the **lib** directory. Its JAR package dependencies are stored in **\testSDK\api-client-test_lib**. You can also download the JAR packages from the **maven** repository.



commons-beanutils-1.8.0.jar	2018/7/20 17:17	Executable Jar File	226 KB
commons-collections-3.2.1.jar	2018/7/20 17:17	Executable Jar File	562 KB
commons-lang-2.5.jar	2018/7/20 17:17	Executable Jar File	273 KB
ezmorph-1.0.6.jar	2018/7/20 17:17	Executable Jar File	85 KB
httpclient-4.5.2.jar	2018/7/20 17:17	Executable Jar File	720 KB
httpcore-4.4.4.jar	2018/7/20 17:17	Executable Jar File	320 KB
jackson-annotations-2.5.4.jar	2018/7/20 17:17	Executable Jar File	39 KB
jackson-core-2.5.4.jar	2018/7/20 17:17	Executable Jar File	225 KB
jackson-databind-2.5.4.jar	2018/7/20 17:17	Executable Jar File	1,118 KB
jcl-over-slf4j-1.7.25.jar	2018/7/20 17:17	Executable Jar File	17 KB
json-lib-2.4.jar	2018/7/20 17:17	Executable Jar File	156 KB
logback-classic-1.1.11.jar	2018/7/20 17:17	Executable Jar File	302 KB
logback-core-1.1.11.jar	2018/7/20 17:17	Executable Jar File	465 KB
slf4j-api-1.7.22.jar	2018/7/20 17:17	Executable Jar File	41 KB

- The JAR package dependencies for the Java SDK Demo are stored in the **components** folder. You can also download the JAR packages from the **maven** repository.



httpmime-4.5.2.jar
json-lib-2.4.jar
netty-all-4.0.27.Final.jar
spring-boot-starter-web-1.5.9.RELEASE.jar

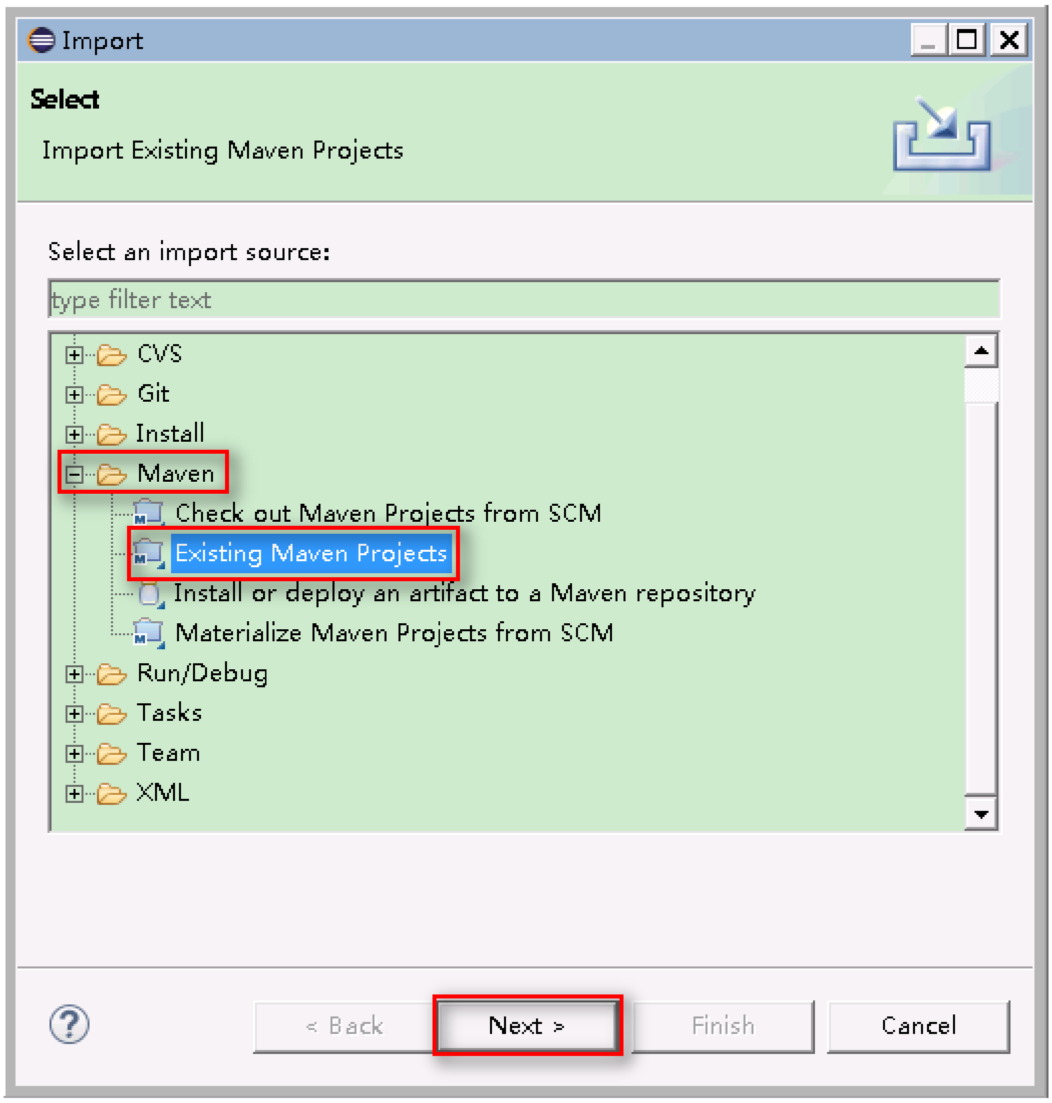
NOTE

The **lib** directory in the Java SDK Demo contains the SDK library.

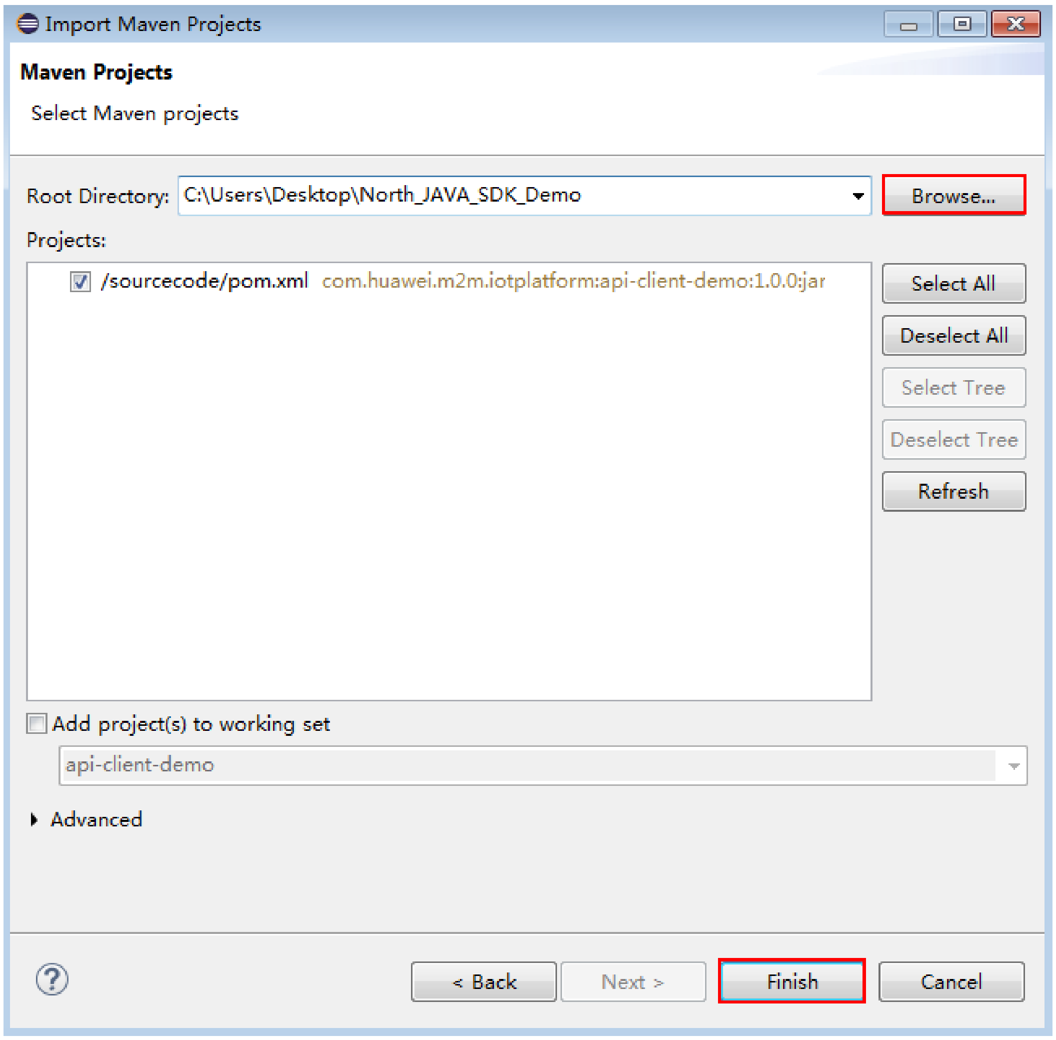
5.1.4 Importing the Java SDK Demo

Step 1 Decompress the downloaded package **OceanConJavaDemo.zip** to a local directory.

Step 2 Start Eclipse, choose **File** > **Import** > **Maven** > **Existing Maven Projects**, and click **Next**.


Import
Select
Import Existing Maven Projects
Select an import source:
type filter text
CVS
Git
Install
Maven
Check out Maven Projects from SCM
Existing Maven Projects
Install or deploy an artifact to a Maven repository
Materialize Maven Projects from SCM
Run/Debug
Tasks
Team
XML
< Back
Next >
Finish
Cancel

Step 3 Click **Browse**. Select the path to which the Java SDK Demo package is decompressed. Then, click **Finish**.


Import Maven Projects

Maven Projects

Select Maven projects

Root Directory: C:\Users\Desktop\North_JAVA_SDK_Demo

Projects:

[✓] /sourcecode/pom.xml com.huawei.m2m.iotplatform:api-client-demo:1.0.0:jar

[ ] Add project(s) to working set

api-client-demo

----End

5.1.5 Initializing and Configuring Certificates

Create a **NorthApiClient** instance. Specify **ClientInfo** (including the IoT platform IP address, port number, application ID, and secret) to initialize the certificate.

NOTICE

- In this example, the IoT platform IP address, port number, application ID, and secret are read from the configuration file **./src/main/resources/application.properties**. Therefore, when the values change, you only need to modify the configuration file.
- The certificate mentioned in this section is provided by the IoT platform for use when calling related APIs. Generally, this certificate is different from the one used for API callback.

Using a Test Certificate

If the test certificate is used:

```
NorthApiClient northApiClient = new NorthApiClient();

PropertyUtil.init("./src/main/resources/application.properties");

ClientInfo clientInfo = new ClientInfo();
clientInfo.setPlatformIp(PropertyUtil.getProperty("platformIp"));
clientInfo.setPlatformPort(PropertyUtil.getProperty("platformPort"));
clientInfo.setAppId(PropertyUtil.getProperty("appId"));
clientInfo.setSecret(PropertyUtil.getProperty("secret"));

northApiClient.setClientInfo(clientInfo);
northApiClient.initSSLConfig();//The default certificate is a test certificate.
The host name is not verified.
```

Using a Specified Certificate

If the test certificate is not used, you can manually specify a certificate (for example, a commercial certificate).

```
NorthApiClient northApiClient = new NorthApiClient();

PropertyUtil.init("./src/main/resources/application.properties");

ClientInfo clientInfo = new ClientInfo();
clientInfo.setPlatformIp(PropertyUtil.getProperty("platformIp"));
clientInfo.setPlatformPort(PropertyUtil.getProperty("platformPort"));
clientInfo.setAppId(PropertyUtil.getProperty("appId"));
clientInfo.setSecret(getAesPropertyValue("secret"));

SSLConfig sslConfig= new SSLConfig();
sslConfig.setTrustCAPath(PropertyUtil.getProperty("newCaFile"));
slConfig.setTrustCAPwd(getAesPropertyValue("newCaPassword"));
slConfig.setSelfCertPath(PropertyUtil.getProperty("newClientCertFile"));
slConfig.setSelfCertPwd(getAesPropertyValue("newClientCertPassword"));

northApiClient.setClientInfo(clientInfo);
northApiClient.initSSLConfig(sslconfig); //Use the specified certificate. Strict
host name verification is used by default.
```

If strict host name verification is not used when a specified certificate is used, you can define the host name verification method before calling **northApiClient.initSSLConfig(sslconfig)**.

```
northApiClient.setHostnameVerifier(new HostnameVerifier() {
    public boolean verify(String arg0, SSLSession arg1) {
      //Customized host name verification
        ......
        return true;
    }
});
```

NOTICE

The method for host name verification should follow security-first principles. The value **true** should not be returned directly.

5.1.6 Calling Service APIs

You can call other service APIs only after the **NorthApiClient** instance is configured by following the instructions provided in **Initializing and Configuring Certificates**. The following APIs are used as an example to describe how to call service APIs:

Authentication

```
//After the NorthApiClient instance is obtained, use the NorthApiClient to obtain
the authentication instance.
Authentication authentication = new Authentication(northApiClient);

//Call the service API provided by the authentication instance, for example,
getAuthToken.
AuthOutDTO authOutDTO = authentication.getAuthToken();

//Obtain the required parameters from the returned authOutDTO, for example,
accessToken.
String accessToken = authOutDTO.getAccessToken();
```

Subscription

```
//After the NorthApiClient instance is obtained, use the NorthApiClient to obtain
the subscription instance.
SubscriptionManagement subscriptionManagement = new
SubscriptionManagement(northApiClient);

//Set the first input parameter SubDeviceDataInDTO in the subDeviceData API.
SubDeviceDataInDTO sddInDTO = new SubDeviceDataInDTO();
sddInDTO.setNotifyType("deviceDataChanged");
//Modify the callback IP address and port number based on site requirements.
ddInDTO.setCallbackUrl("https://XXX.XXX.XXX.XXX:8099/v1.0.0/messageReceiver");
try {
   //Call the service API provided by the subscription class instance
subscriptionManagement, for example, subDeviceData.
    SubscriptionDTO subDTO = subscriptionManagement.subDeviceData(sddInDTO, null,
accessToken);
    System.out.println(subDTO.toString());
} catch (NorthApiException e) {
    System.out.println(e.toString());
}
```

Device Registration

```
//After the NorthApiClient instance is obtained, use the NorthApiClient to obtain
the device management instance.
DeviceManagement deviceManagement = new DeviceManagement(northApiClient);

//Set the first input parameter RegDirectDeviceInDTO2 in the regDirectDevice API.
RegDirectDeviceInDTO2 rddInDTO = new RegDirectDeviceInDTO2();
String nodeid = "86370303XXXXXX"; //this is a test imei
String verifyCode = nodeid;
rddInDTO.setNodeId(nodeid);
rddInDTO.setVerifyCode(verifyCode);
rddInDTO.setTimeout(timeout);

//Call the service API provided by the device management instance
deviceManagement, for example, regDirectDevice.
RegDirectDeviceOutDTO rddod = deviceManagement.regDirectDevice(rddInDTO, null,
accessToken);

//Obtain the required parameters from the returned rddod structure, for example,
deviceId.
String deviceId = rddod.getDeviceId();
```

NOTE

For details about mandatory parameters, see *Northbound Java SDK API Reference*. If a parameter is not required, it can be left empty or set to **null**.

5.1.7 Implementing Callback APIs and Making, Exporting, and Uploading a Callback Certificate

Implementing Callback APIs

Create a class inheriting **PushMessageReceiver**. If a specific type of message needs to be received, overwrite the corresponding method. For details, see **PushMessageReceiverTest** in the Java SDK Demo.

```
@Override
public void handleDeviceAdded(NotifyDeviceAddedDTO body) {
    System.out.println("deviceAdded ==> " + body);
    //TODO deal with deviceAdded notification
}
```

NOTE

- A message pushed by the IoT platform must be processed according to the service. However, complex calculations, I/O operations, and operations that will take a long time are not recommended. You can write data into the database, and access or refresh the corresponding page before obtaining data from the database.
- The callback path has been set in the SDK. Therefore, pay attention to the callback URL during subscription. For details, see APIs in the "Message Push" section in the *Java SDK API Reference Document*.
- The callback IP address is the same as that of the server. It must be a public IP address.
- The callback port of the Java SDK Demo is configured in the **src\main\resource\application.properties** directory.

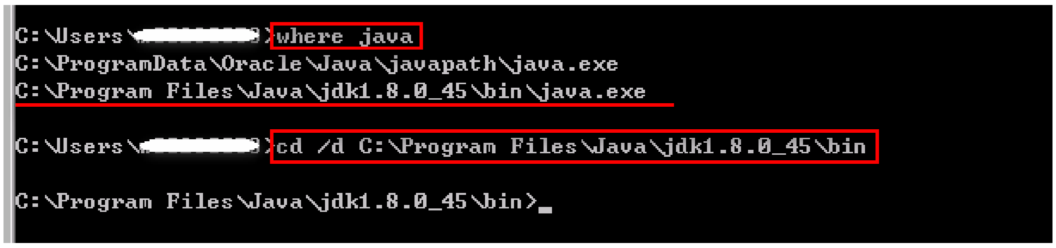
  ```
  #specify the port of the web application
  server.port=8099
  ```

Making a Callback Certificate

A self-signed certificate is used as an example. A commercial certificate must be applied from the CA.

Step 1 Open the Windows CLI, and enter **where java** to switch to the **bin** directory of the JDK.

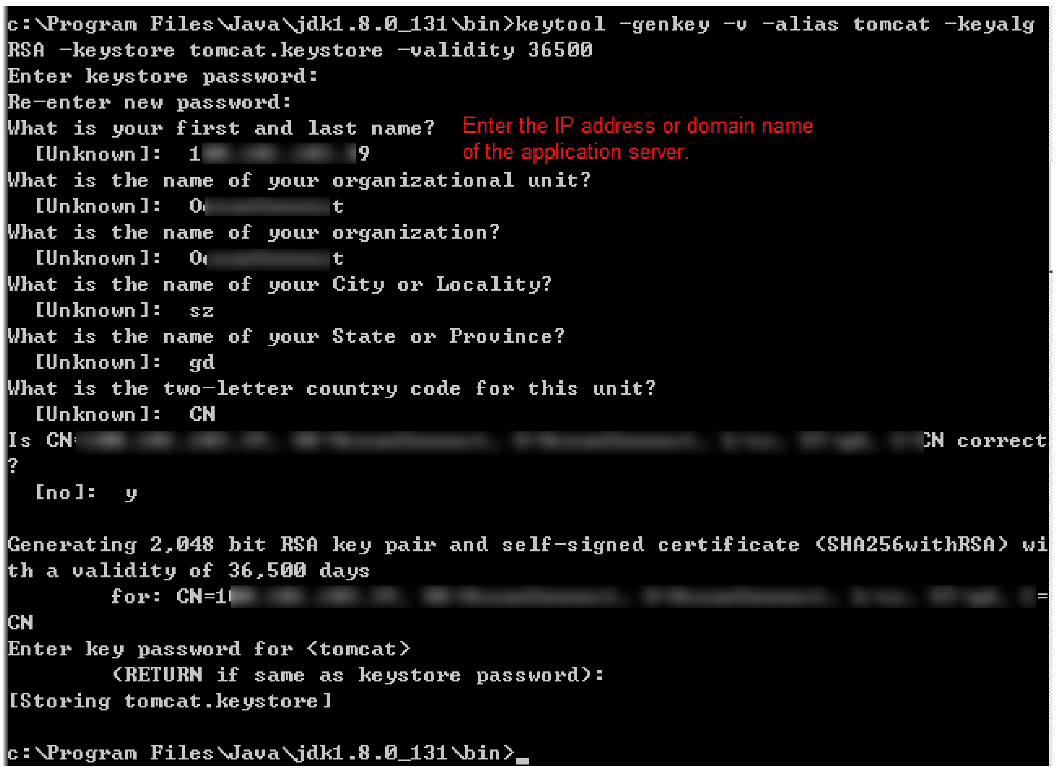
```
where java
Cd /d {bin directory of the JDK}
```



```
C:\Users\[redacted]>where java
C:\ProgramData\Oracle\Java\javapath\java.exe
C:\Program Files\Java\jdk1.8.0_45\bin\java.exe

C:\Users\[redacted]>cd /d C:\Program Files\Java\jdk1.8.0_45\bin

C:\Program Files\Java\jdk1.8.0_45\bin>_
```

Step 2 Run the following command to generate the **tomcat.keystore** file:

```
keytool -genkey -v -alias tomcat -keyalg RSA -keystore tomcat.keystore -validity
36500
```


Enter the IP address or domain name
of the application server.

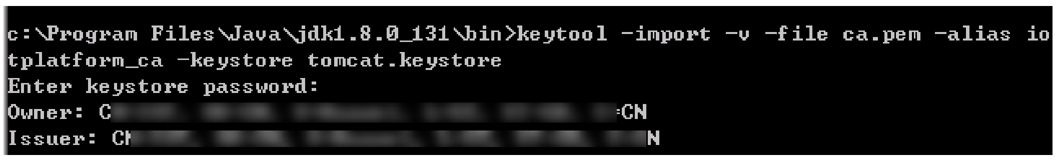
```
c:\Program Files\Java\jdk1.8.0_131\bin>keytool -genkey -v -alias tomcat -keyalg
RSA -keystore tomcat.keystore -validity 36500
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  1[redacted]9
What is the name of your organizational unit?
  [Unknown]:  O[redacted]t
What is the name of your organization?
  [Unknown]:  O[redacted]t
What is the name of your City or Locality?
  [Unknown]:  sz
What is the name of your State or Province?
  [Unknown]:  gd
What is the two-letter country code for this unit?
  [Unknown]:  CN
Is CN[redacted]CN correct
?
  [no]:  y

Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA) wi
th a validity of 36,500 days
        for: CN=1[redacted]=
CN
Enter key password for <tomcat>
        (RETURN if same as keystore password):
[Storing tomcat.keystore]

c:\Program Files\Java\jdk1.8.0_131\bin>_
```

NOTICE

- If the **tomcat.keystore** file exists in the **bin** directory of the JDK, move the file to another path.
- Enter the IP address or domain name of the application server under **What is your first and last name?**
- The password of **<tomcat>** must be the same as that of the keystore (press **Enter** in the last step). **Remember the password of the keystore, as it will be used in subsequent configurations**.

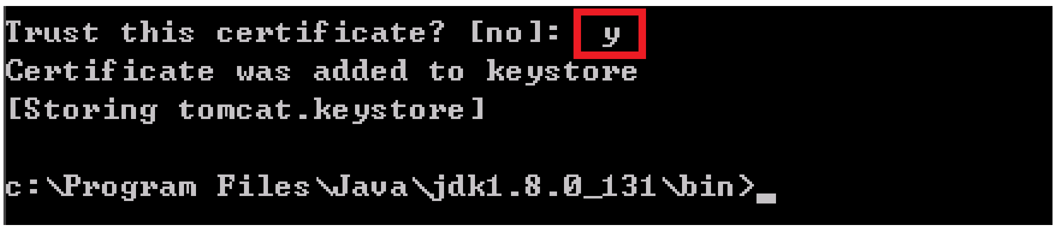
Step 3 Place the root certificate **ca.pem** provided by the IoT platform in the **bin** directory of the JDK and run the following command to add it to the trust certificate chain of the **tomcat.keystore** file:

```
keytool -import -v -file ca.pem -alias iotplatform_ca -keystore tomcat.keystore
```



```
c:\Program Files\Java\jdk1.8.0_131\bin>keytool -import -v -file ca.pem -alias io
tplatform_ca -keystore tomcat.keystore
Enter keystore password:
Owner: C[redacted]CN
Issuer: C[redacted]N
```

Enter the keystore password. Check the imported certificate content and enter **y**.



```
Trust this certificate? [no]:  y
Certificate was added to keystore
[Storing tomcat.keystore]

c:\Program Files\Java\jdk1.8.0_131\bin>_
```

NOTE

- The test root certificate **ca.pem** of the IoT platform can be found in the **cert** directory of the Java SDK package.
- After the root certificate **ca.pem** provided by the IoT platform is added to the trust certificate chain of the **tomcat.keystore** file, the sub-certificate issued by **ca.pem** can obtain the trust of the application server.

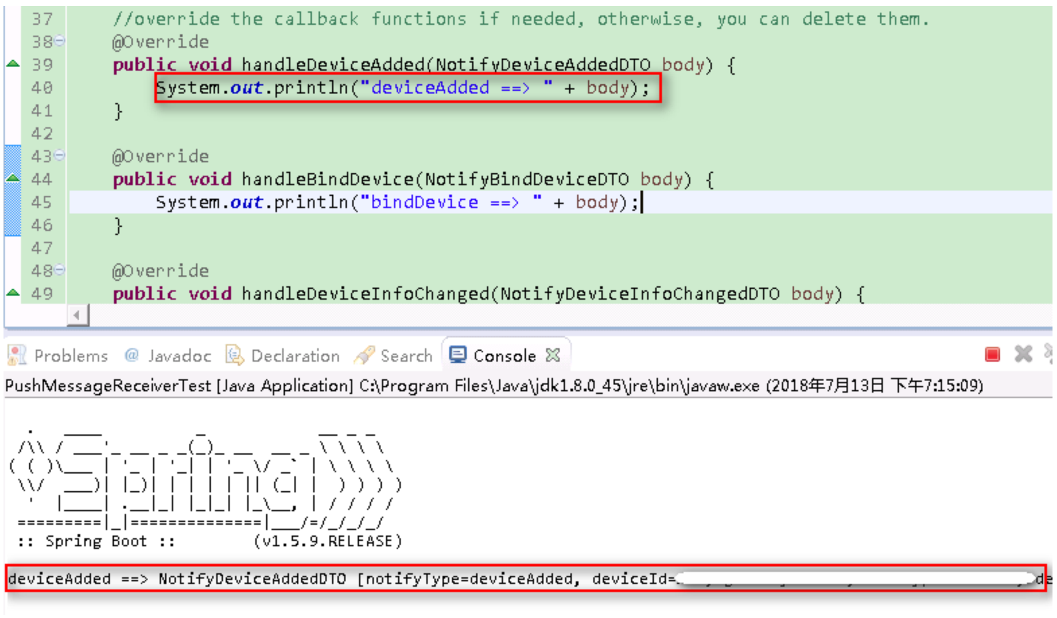
Step 4 Place the **tomcat.keystore** file in the directory of the Java SDK Demo, for example, **src\main\resources**. Open the **src\main\resource\application.properties** file and add the following configuration, where **server.ssl.key-store** indicates the path where the **tomcat.keystore** file is stored and **server.ssl.key-store-password** indicates the password of the keystore:

```
#one-way authentication (server-auth)
server.ssl.key-store=./src/main/resources/tomcat.keystore
server.ssl.key-store-password=741852963.
```

Step 5 Right-click **PushMessageReceiverTest** and choose **Run As** > **Java Application** to run the **PushMessageReceiverTest** class in the Java SDK Demo. The command output is as follows:

NOTE

Data is transferred to the corresponding callback function when it is pushed to the application server.



```
    //override the callback functions if needed, otherwise, you can delete them.
    @Override
    public void handleDeviceAdded(NotifyDeviceAddedDTO body) {
        System.out.println("deviceAdded ==> " + body);
    }

    @Override
    public void handleBindDevice(NotifyBindDeviceDTO body) {
        System.out.println("bindDevice ==> " + body);
    }

    @Override
    public void handleDeviceInfoChanged(NotifyDeviceInfoChangedDTO body) {
```

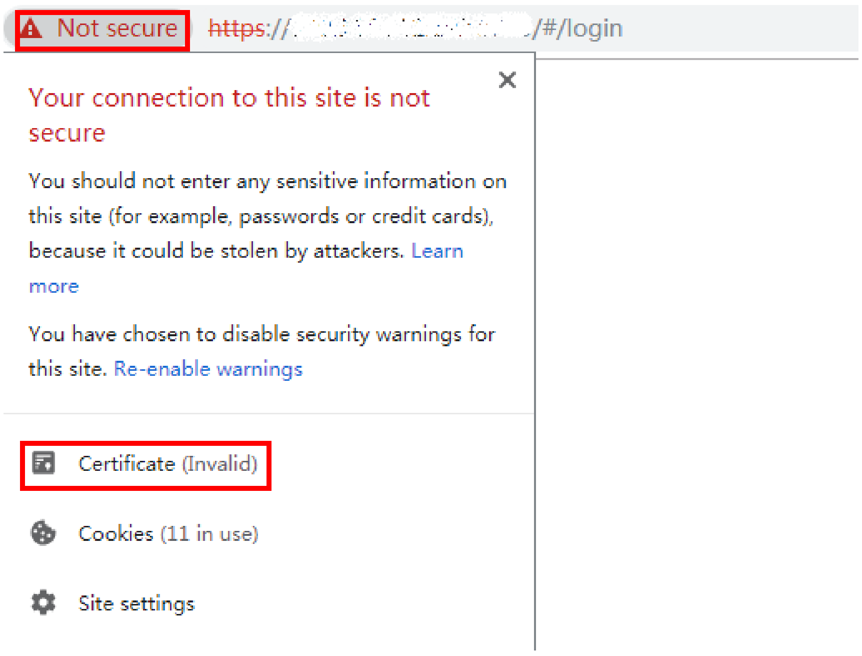
```
PushMessageReceiverTest [Java Application] C:\Program Files\Java\jdk1.8.0_45\jre\bin\javaw.exe (2018年7月13日 下午7:15:09)

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.5.9.RELEASE)

deviceAdded ==> NotifyDeviceAddedDTO [notifyType=deviceAdded, deviceId=[redacted]de
```

----End

Exporting a Callback Certificate

Step 1 Use a browser to open the callback URL **https://server:8099/v1.0.0/messageReceiver** and view the certificate. Google Chrome is used as an example.

The IP address of the server is the same as that of the local host. 8099 is the port configured in the **application.properties** file.



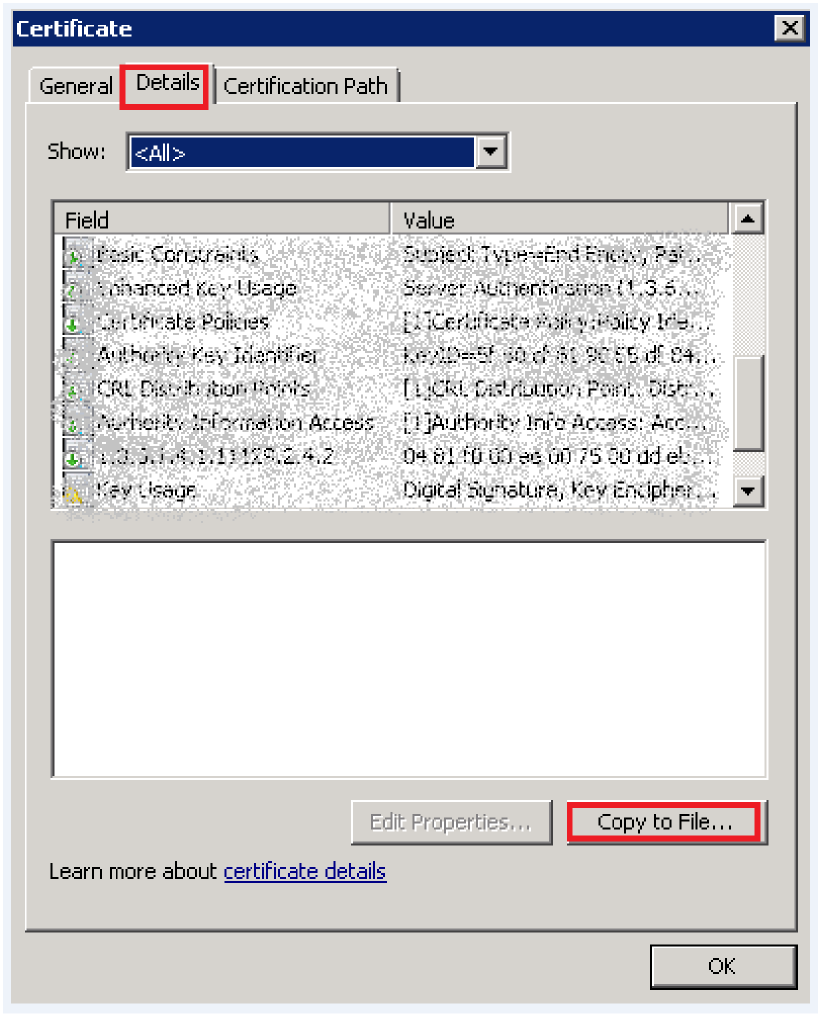
Not secure   https://[illegible]/#/login

Your connection to this site is not secure

You should not enter any sensitive information on this site (for example, passwords or credit cards), because it could be stolen by attackers. Learn more

You have chosen to disable security warnings for this site. Re-enable warnings

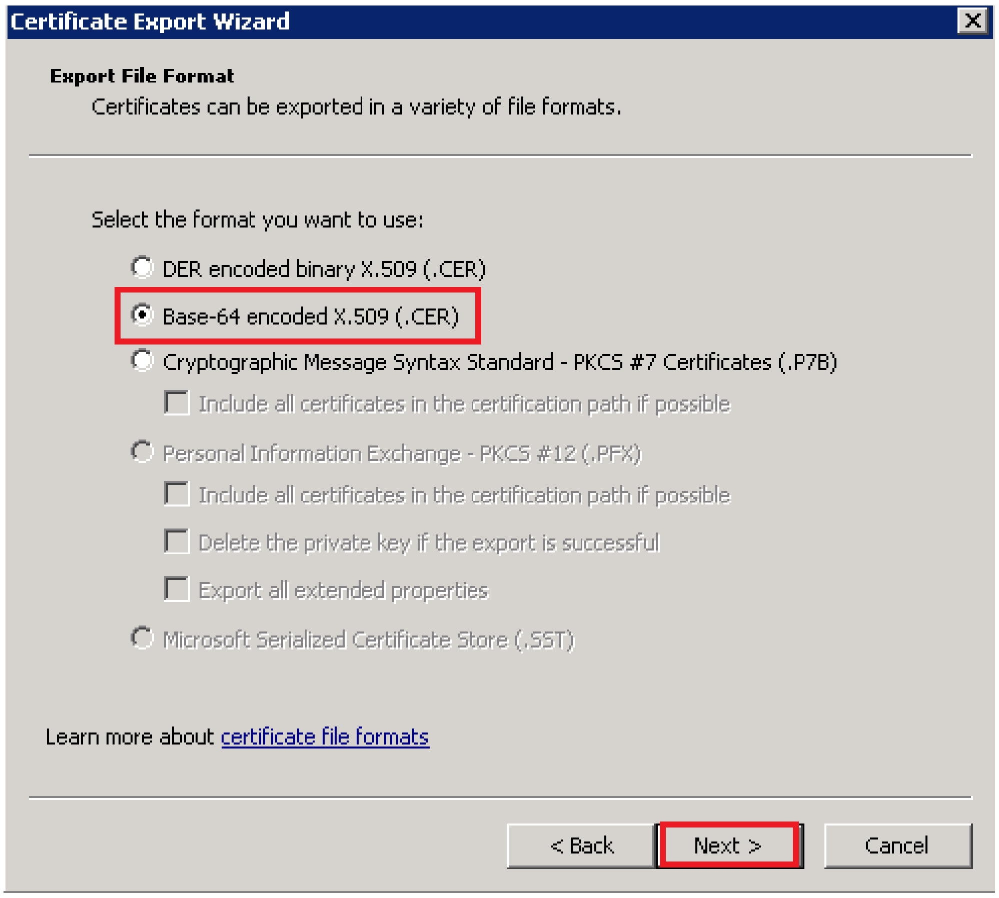
Step 2 In the **Certificate** dialog box, click the **Details** tab. Click **Copy to File**.



Show: <All>

Field	Value
Basic Constraints	Subject Type=End Entity, Pat...
Enhanced Key Usage	Server Authentication (1.3.6...
Certificate Policies	[1]Certificate Policy:Policy Ide...
Authority Key Identifier	KeyID=5f 90 cf 81 90 55 df 04...
CRL Distribution Points	[1]CRL Distribution Point: Distr...
Authority Information Access	[1]Authority Info Access: Acc...
1.3.6.1.4.1.11129.2.4.2	04 81 f0 00 ee 00 75 00 dd eb...
Key Usage	Digital Signature, Key Encipher...

Step 3 Click **Next**. In the **Export File Format** dialog box displayed, select **Base-64 encoded X.509 (.CER)**, and click **Next**.


Certificate Export Wizard

Export File Format
Certificates can be exported in a variety of file formats.

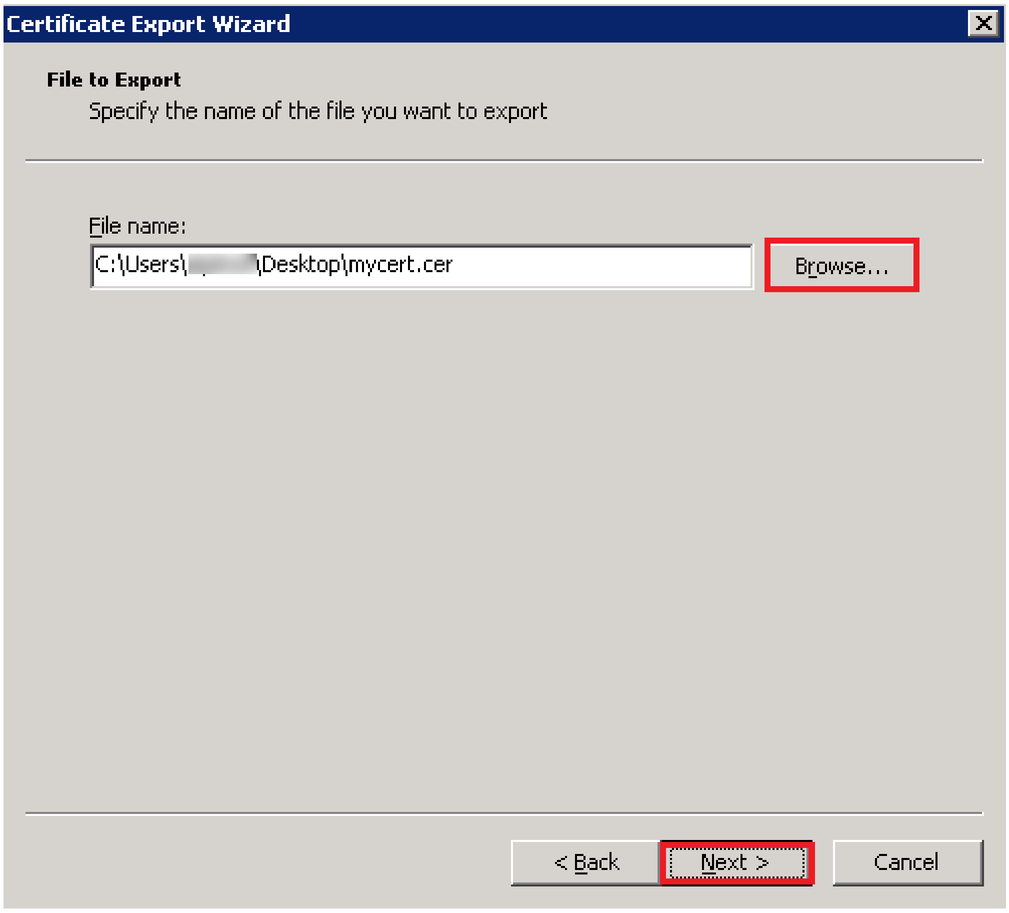
Select the format you want to use:

- ( ) DER encoded binary X.509 (.CER)
- (●) Base-64 encoded X.509 (.CER)
- ( ) Cryptographic Message Syntax Standard - PKCS #7 Certificates (.P7B)
  - ☐ Include all certificates in the certification path if possible
- ( ) Personal Information Exchange - PKCS #12 (.PFX)
  - ☐ Include all certificates in the certification path if possible
  - ☐ Delete the private key if the export is successful
  - ☐ Export all extended properties
- ( ) Microsoft Serialized Certificate Store (.SST)

Learn more about certificate file formats

Step 4 Specify the path for saving the certificate.

1. In the **File to Export** dialog box, click **Browse**. Select a path, enter the file name, and click **Save** to return to the **Certificate Export Wizard** dialog box. Then, click **Next**.


Certificate Export Wizard

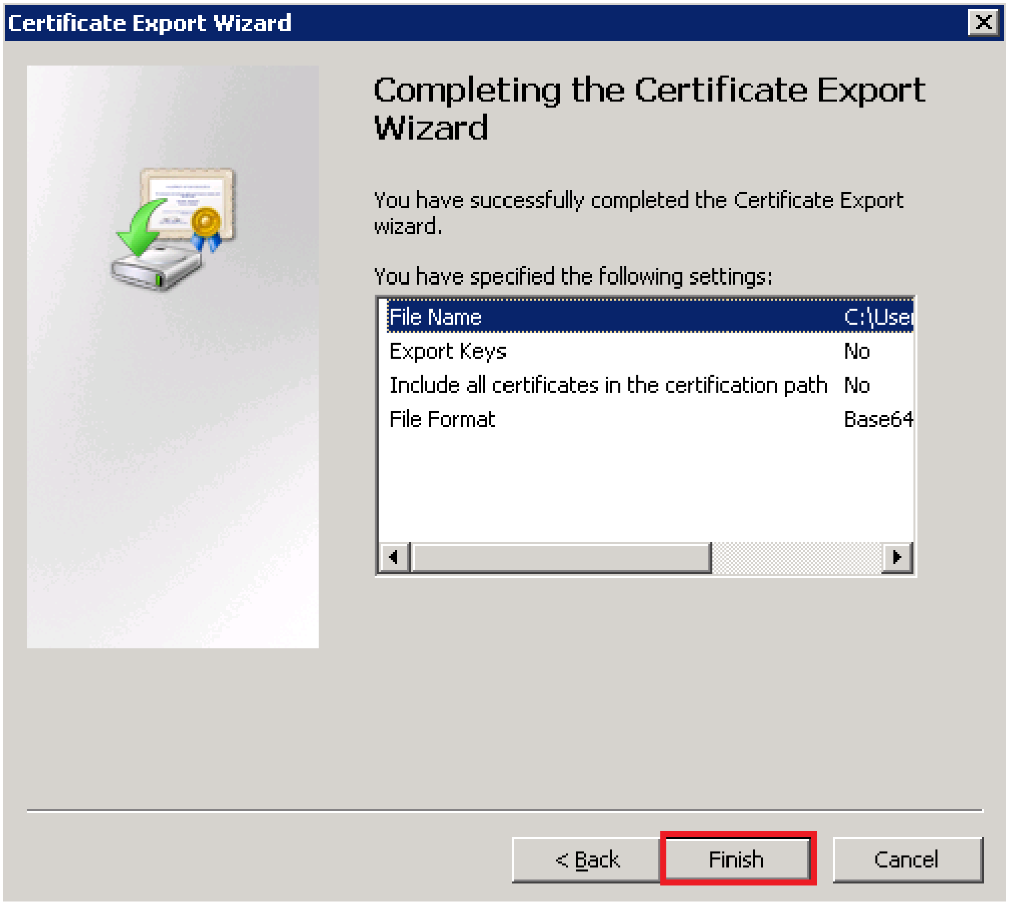
File to Export

Specify the name of the file you want to export

File name:

C:\Users\[redacted]\Desktop\mycert.cer

2. Click **Finish** to export the certificate.



Completing the Certificate Export Wizard

You have successfully completed the Certificate Export wizard.

You have specified the following settings:

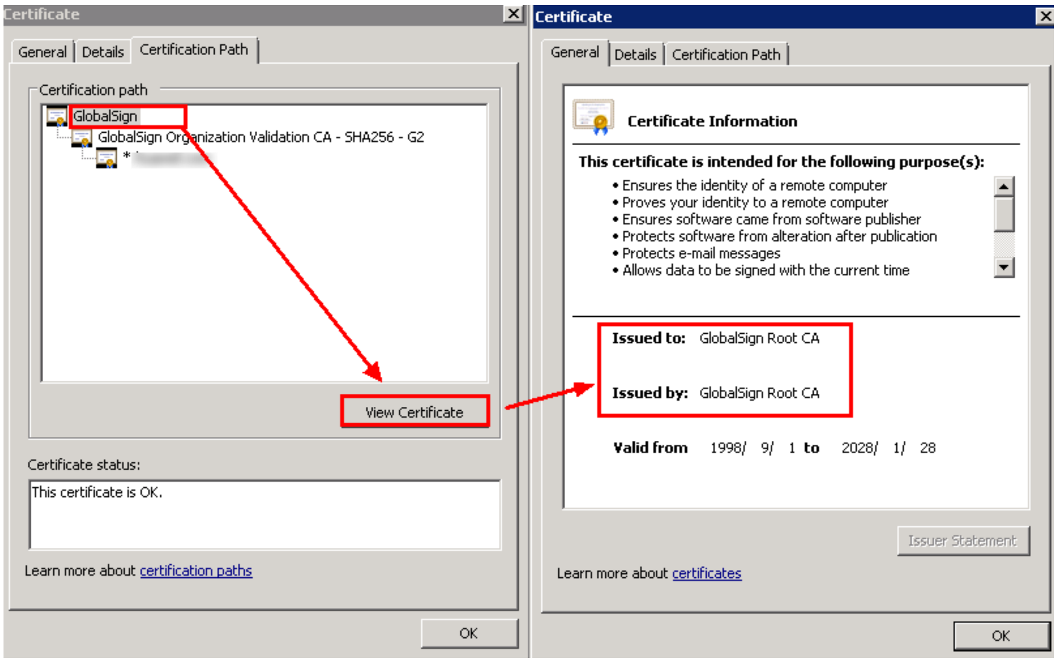
File Name	C:\Use
Export Keys	No
Include all certificates in the certification path	No
File Format	Base64

NOTICE

If the application server is deployed on the cloud, multiple certificates may exist. **You are advised to export the certificates one by one after the deployment is complete**.

Step 5 If multi-level certificates exist, export them one by one.

1. In the **Certificate** dialog box, select a certificate path and click **View Certificate**.


Certificate
General
Details
Certification Path
Certification path
GlobalSign
GlobalSign Organization Validation CA - SHA256 - G2
View Certificate
Certificate status:
This certificate is OK.
Learn more about certification paths
OK
Certificate
Certificate Information
This certificate is intended for the following purpose(s):
• Ensures the identity of a remote computer
• Proves your identity to a remote computer
• Ensures software came from software publisher
• Protects software from alteration after publication
• Protects e-mail messages
• Allows data to be signed with the current time
Issued to: GlobalSign Root CA
Issued by: GlobalSign Root CA
Valid from 1998/ 9/ 1 to 2028/ 1/ 28
Issuer Statement
Learn more about certificates
OK

2. Click the **Details** tab and repeat the preceding steps to export every selected certificate.

Step 6 Use the text editor to combine all the exported certificates sequentially into a PEM file. This file must be uploaded to the corresponding application on the IoT platform.

```
5gaRQBi5+MHt39tBquCWIMnNZBU4gcmU7qKEKQsTb47bDN0lAtukixlE0kF6BWlK
WE9gyn6CagsCqiUXObXbf+eEZSqVir2G3l6BFoMtEMze/aiCKm0oHw0LxOXnGiYZ
4fQRbxC1lfznQgUy286dUV4otp6F01vvpX1FQHKOtw5rDgb7MzVIcbidJ4vEZV8N
hnacRHr2lVz2XTIIM6RUthg/aFzyQkqFOFSDX9HoLPKsEdao7WNq
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFODCCBCCgAwIBAgIQUT+5dDhwtzRAQY0wkwaZ/zANBgkqhkiG9w0BAQsFADCB
yjELMAkGA1UEBhMCVVMxFzAVBgNVBAoTDlZlcmlTaWduLCBJbmMuMR8wHQYDVQQL
ExZWZXJpU2lnbiBUcnVzdCBOZXR3b3JrMTowOAYDVQQLEzEoYykgMjAwNiBWZXJp
U2lnbiwgSW5jLiAtIEZvciBhdXRob3JpemVkIHVzZSBvbmx5MUUwQwYDVQQDEzxW
ZXJpU2lnbiBDbGFzcyAzIFB1YmxpYyBQcmltYXJ5IENlcnRpZmljYXRpb24gQXV0
```

----End

NOTICE

- The configuration in the Demo is one-way authentication. After the certificate is exported, change it to two-way authentication. Open the following configuration file (delete the comment and change the tomcat.keystore directory and password). The root certificate of the platform has been added to the tomcat.keystore trust certificate chain. Therefore, you do not need to modify the configuration file. Restart the server.

```
#two-way authentication (add client-auth)
server.ssl.trust-store=./src/main/resources/tomcat.keystore
server.ssl.trust-store-password=741852963.
server.ssl.client-auth=need
```

- One-way authentication is less secure than two-way authentication. Therefore, two-way authentication is recommended.

Uploading the Callback Certificate

Step 1 Log in to the Developer Center and access a project.

Step 2 Choose **Applications** > **Interconnection**, and click **Certificate Management**.

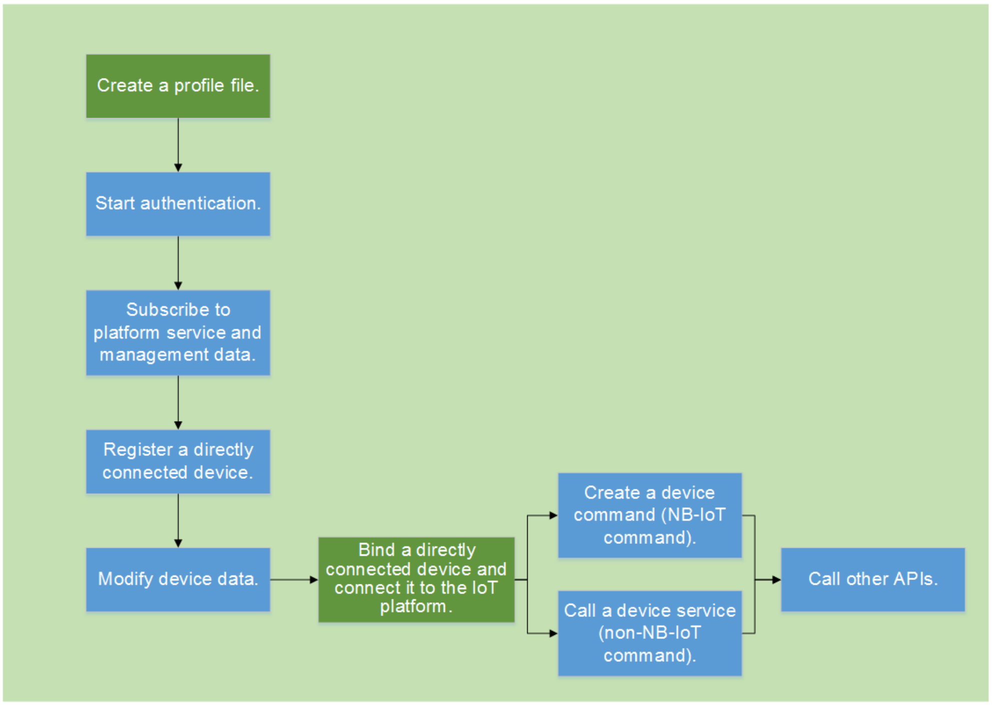
Step 3 Click **Add** to upload the certificate.

----End

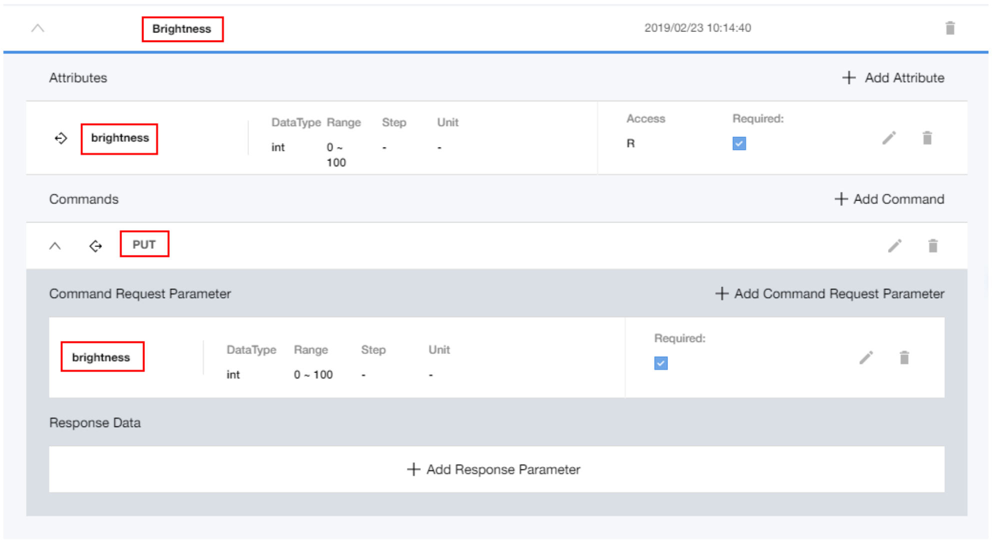
5.1.8 Service API Calling Process and Precautions

The methods for calling other APIs are similar to those for calling service APIs. For details, see **Calling Service APIs**.

- The following figure shows the flow for calling a service API.


Create a profile file.
Start authentication.
Subscribe to platform service and management data.
Register a directly connected device.
Modify device data.
Bind a directly connected device and connect it to the IoT platform.
Create a device command (NB-IoT command).
Call a device service (non-NB-IoT command).
Call other APIs.

- The following figure shows the profile file used in the Java SDK Demo. There is only one Brightness service, which contains a brightness attribute and a PUT command. **When calling a device command or device service API, change the service, attribute, or command name to the corresponding name if the following profile content is not used.**



	DataType	Range	Step	Unit	Access	Required:
brightness	int	0 ~ 100	-	-	R	✓

	DataType	Range	Step	Unit	Required:
brightness	int	0 ~ 100	-	-	✓

- To create a profile, perform the following steps:

  Log in to the Developer Center, choose **Product** > **Product Development** > **Add** > **Customization**, and click **Customization** to open the **Set Product Information** page. Specify **Product Name**, **Model**, **Manufacture ID**, **Industry**, **Device Type**, and **Protocol Type**, and click **Create**. Click **+Add Service** to add attributes and commands based on device functions, and click **Save**.

  NOTE

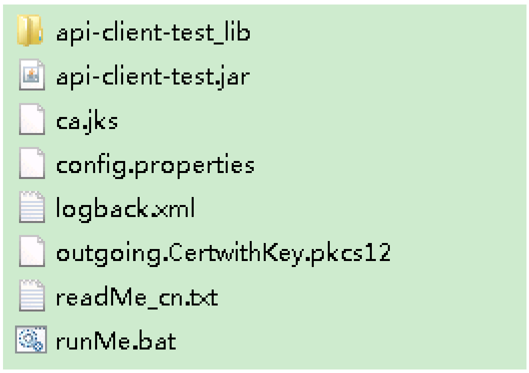
  You are advised to call the API to register the device after the profile file is defined.

- The values of **DeviceType**, **ManufacturerId**, **ManufacturerName**, and **Model** must be the same as those defined in the profile file.
- The accessToken can be managed by the SDK or third-party applications. For details, choose **Secure Application Access** > **Periodically Refreshing a Token** in the *Northbound Java SDK API Reference*.

5.1.9 Testing the SDK

The SDK packages provide JAR packages that can be run independently to test the related northbound APIs provided by the IoT platform. JAR packages that can run independently are stored in the **testSDK** directory.

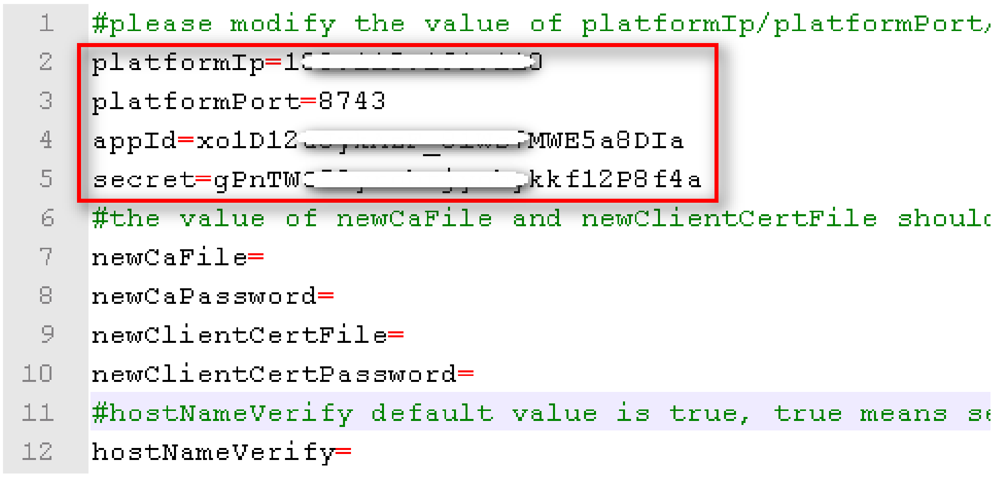
Figure 5-1 JAR packages that can be run independently



api-client-test_lib
api-client-test.jar
ca.jks
config.properties
logback.xml
outgoing.CertwithKey.pkcs12
readMe_cn.txt
runMe.bat

Step 1 Modify the **config.properties** file and double-click **runMe.bat** to perform the test.

Figure 5-2 Modifying config.properties



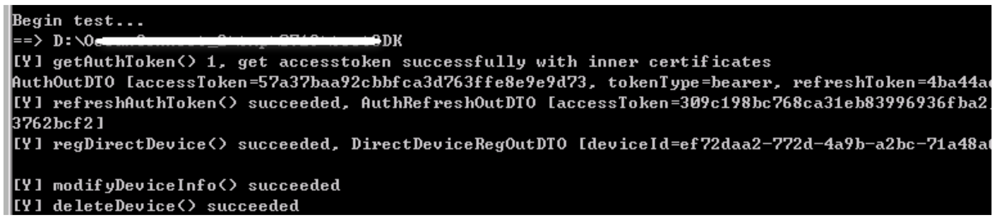
```
#please modify the value of platformIp/platformPort,
platformIp=1[redacted]3
platformPort=8743
appId=xo1D12[redacted]MWE5a8DIa
secret=gPnTW[redacted]kkf12P8f4a
#the value of newCaFile and newClientCertFile shoulc
newCaFile=
newCaPassword=
newClientCertFile=
newClientCertPassword=
#hostNameVerify default value is true, true means se
hostNameVerify=
```

Step 2 If a commercial certificate is used, place it in the **testSDK** directory (the certificate name cannot be **ca.jks** or **outgoing.CertwithKey.pkcs12**) and configure the certificate name and password in the **config.properties** file. If the test certificate is used, you do not need to modify the certificate data in the **config.properties** file.

Step 3 The test result is displayed at the beginning. **[y]** indicates that the test is successful. **[x]** indicates an error. Check the error message or description in that line.

Step 4 The JDK is required to run JAR packages. Ensure that the JDK has been installed and the system environment variables have been set.

The command output is as follows:



```
Begin test...
==> D:\O[redacted]DK
[Y] getAuthToken() 1, get accesstoken successfully with inner certificates
AuthOutDTO [accessToken=57a37baa92cbbfca3d763ffe8e9e9d73, tokenType=bearer, refreshToken=4ba44a
[Y] refreshAuthToken() succeeded, AuthRefreshOutDTO [accessToken=309c198bc768ca31eb83996936fba2
3762bcf2]
[Y] regDirectDevice() succeeded, DirectDeviceRegOutDTO [deviceId=ef72daa2-772d-4a9b-a2bc-71a48a

[Y] modifyDeviceInfo() succeeded
[Y] deleteDevice() succeeded
```

----End