IoT Device Access(IoTDA)
25.9.0.SPC002

User Guide

Issue	01
Date	2025-11-30

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Trademarks and Permissions


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

Notice

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

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

Huawei Cloud Computing Technologies Co., Ltd.

Address: Huawei Cloud Data Center Jiaoxinggong Road
Qianzhong Avenue
Gui'an New District
Gui Zhou 550029
People's Republic of China

Website: https://www.huaweicloud.com/intl/en-us/

Contents

1 Getting Started

1.1 Accessing and Using IoTDA

Step 1 Use an account with the IoT Device Access (IoTDA) permissions to log in to ManageOne Operation Portal with a browser.

URL: **https://***Address for accessing ManageOne Operation Portal*. For example, **https://console.demo.com/momaintenancewebsite/uniportal/#/home**.

Step 2 Click  in the upper left corner of the page and choose **IoT** > **IoT Device Access**.

Step 3 (Optional) If IoTDA is not enabled, click **Apply for IoTDA** and contact your enterprise administrator to enable it.

Step 4 Log in to the IoTDA console.

----End

1.2 Connecting MQTT Devices

Using the MQTT Device Simulator

This topic uses a device simulator as an example to describe how to connect devices to IoTDA using the native MQTT protocol. The device simulator is an MQTT client, which enables you to easily verify whether devices can interact with IoTDA to publish or subscribe to messages.

Prerequisites

You have enabled the IoTDA service through your enterprise administrator.

Obtaining Device Access Information

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

Step 1 Log in to the IoTDA console.

Step 2 Choose **Overview** in the navigation pane, view the MQTTS access address and port, and click **Download CA Certificate** to save the server certificate file.

NOTE

Notes on the device access address

- Generally, only IP address access is supported, and the domain name is displayed as **default**.
- For devices that do not support domain name access, use IP addresses instead.

Notes on the security protocol for device access

- For security purposes, non-encrypted ports, such as CoAP and MQTT ports, are disabled by default. To enable them, contact O&M engineers.
- If a secure protocol is specified for connecting a registered device to the platform, a non-encrypted port cannot be used for device access.
- If an insecure protocol is specified for connecting a registered device to the platform, a non-encrypted port can be used for access. The device successfully connected to the platform using an encrypted port cannot be connected again using a non-encrypted port.

----End

Creating a Product

If you have defined a product model (profile), skip this step.

This example shows how to create a product model for reporting battery information. In this profile:

Service ID: **Battery**, which indicates a service related to the battery.

Property: **batteryLevel**, which is a property of the **Battery** service to indicate the battery level.

The procedure is as follows:

Step 1 Log in to the IoTDA console.

Step 2 Choose **Products** in the navigation pane.

Step 3 Click **Create Product** in the upper right corner, set the parameters based on **Table 1-1**, and click **OK**.

Table 1-1 Creating a product

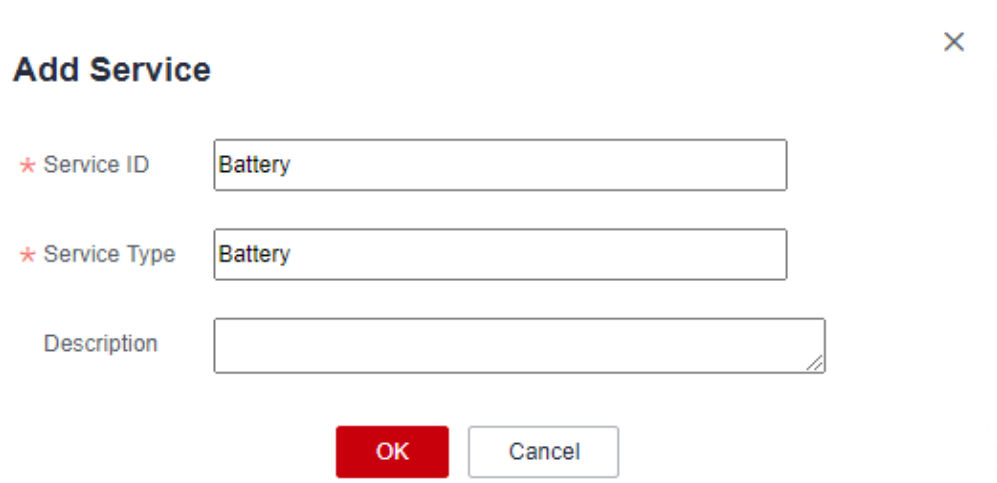
Parameter	Description
Resource Space	IoTDA automatically allocates the created product to the default resource space. • You can select another resource space from the drop-down list box as you want. • You can create a resource space in **Resource Spaces** and allocate your products in it.

Parameter	Description
Product Name	Enter a name unique in the resource space.
Protocol	In this example, select **MQTT**.
Data Type	In this example, select **JSON**. IoTDA communicates with devices in the JSON format, which is a standard format defined by the product model.
Manufacturer	Enter the manufacturer name of the device.
Device Type	You can set it to **StreetLight**, **GasMeter**, or **WaterMeter**.

Step 4 In the row of the product created in **Step 3**, click **View** to access its details.

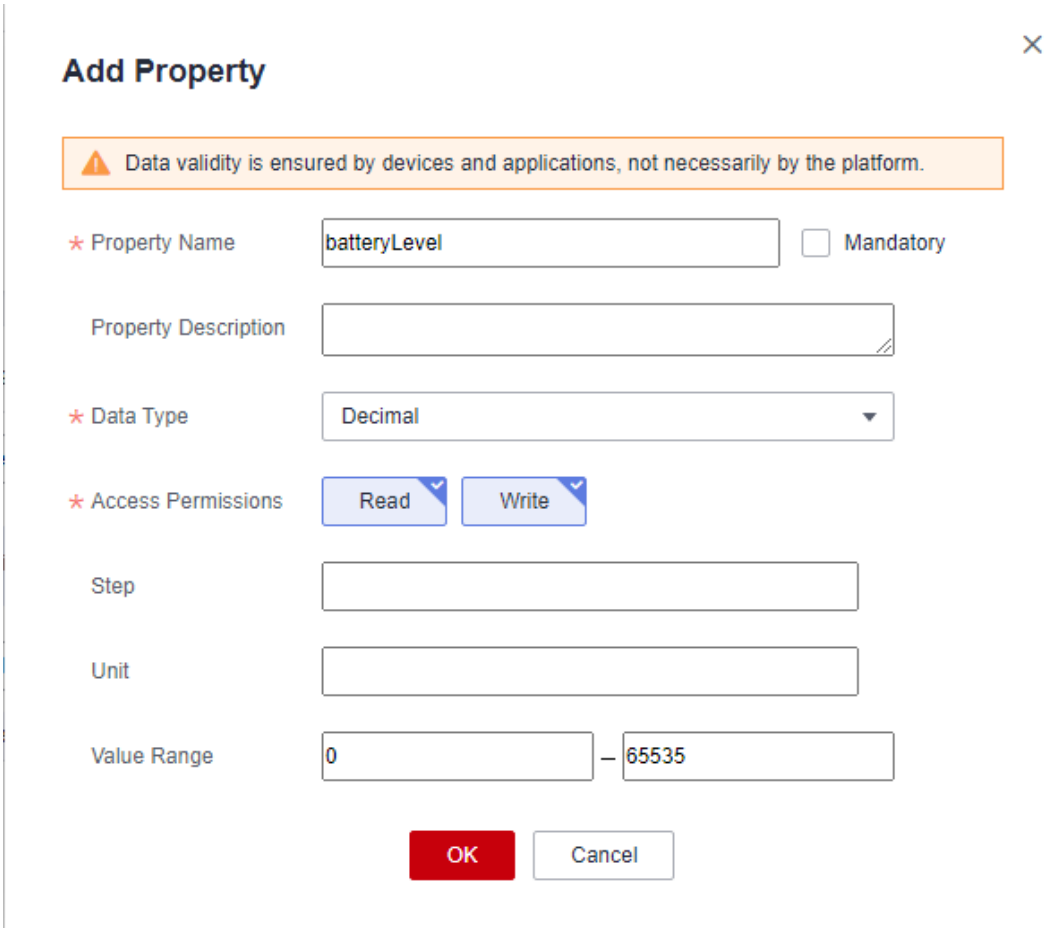
Step 5 On the **Model Definition** tab page, click **Customize Model**, add the **Battery** service, and click **OK**.

Figure 1-1 Adding a service



Step 6 Click the service created in **Step 5**, click **Add Property**, set the parameters as shown in the figure below, and click **OK**.

Figure 1-2 Adding a property



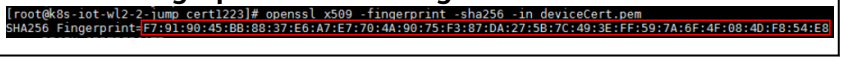
----End

Registering a Device

Step 1 In the navigation pane, choose **Devices** > **All Devices**, click **Register Device** in the upper right corner, set the parameters based on the table below, and click **OK**.

Table 1-2 Registering a device

Parameter	Description
Resource Space	Select the resource space that the device belongs to.
Product	Select the product that the device belongs to. You can select a product only when it is available on the **Products** page. If no product is available, create a product first.
Node ID	Customize a unique physical identifier for the device that uses MQTT for access. The device ID and secret are returned after the registration, and the device uses them to connect to IoTDA.

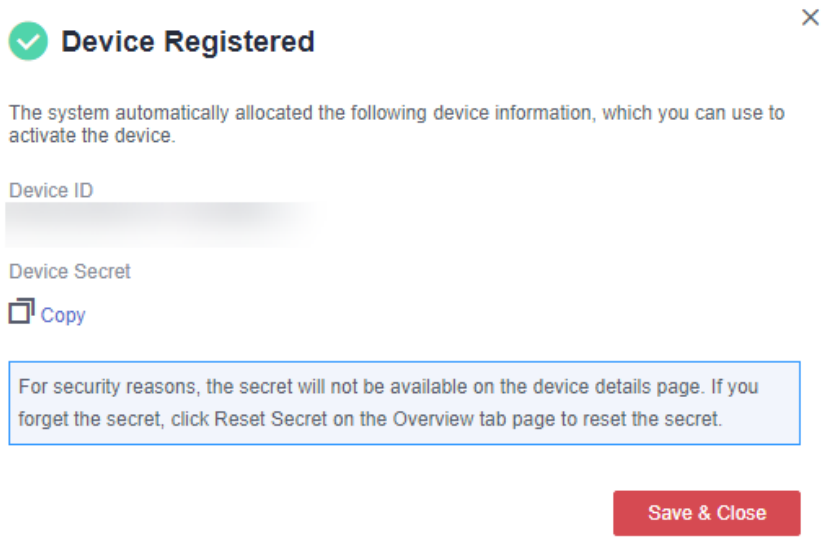
Parameter	Description
Device Name	Enter a custom device name.
Device ID	Enter a unique device ID. If this parameter is carried, the platform will use the parameter value as the device ID. Otherwise, the platform will allocate a device ID, which is in the format of *[product_id]_ [node_id]*.
Device Description	Describe the device.
Authentication Type	• **Secret**: The device uses the secret for identity verification. • **X.509 certificate**: The device uses an X.509 certificate for identity verification. Before registering a device authenticated by an X.509 certificate, upload the device CA certificate to IoTDA and bind the device certificate to the device during device registration. For details, see **Device CA Certificates**.
Secret	Specify a secret used for device access. If no secret is specified, IoTDA automatically generates one.
Confirm Secret	Keep it the same as the entered secret.
Fingerprint	This parameter is displayed when Authentication Type is set to **X.509 certificate**. Enter the fingerprint of **Presetting an X.509 Certificate**. You can run the **openssl x509 -fingerprint -sha256 -in deviceCert.pem** command in the OpenSSL view to query the fingerprint. **Note: Delete the colon (:) from the obtained fingerprint when filling it.** 

Step 2 Save the device ID and secret. They are used for authentication when the device attempts to access IoTDA.

NOTE

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

Figure 1-3 Creating a device



----End

Connecting a Device Simulator to IoTDA

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

Step 2 Open the MQTT.fx client and choose **Extras** > **Edit Connection Profiles** from the menu bar. On the displayed page, set related parameters and click **OK**.

Figure 1-4 MQTT.fx connection parameters

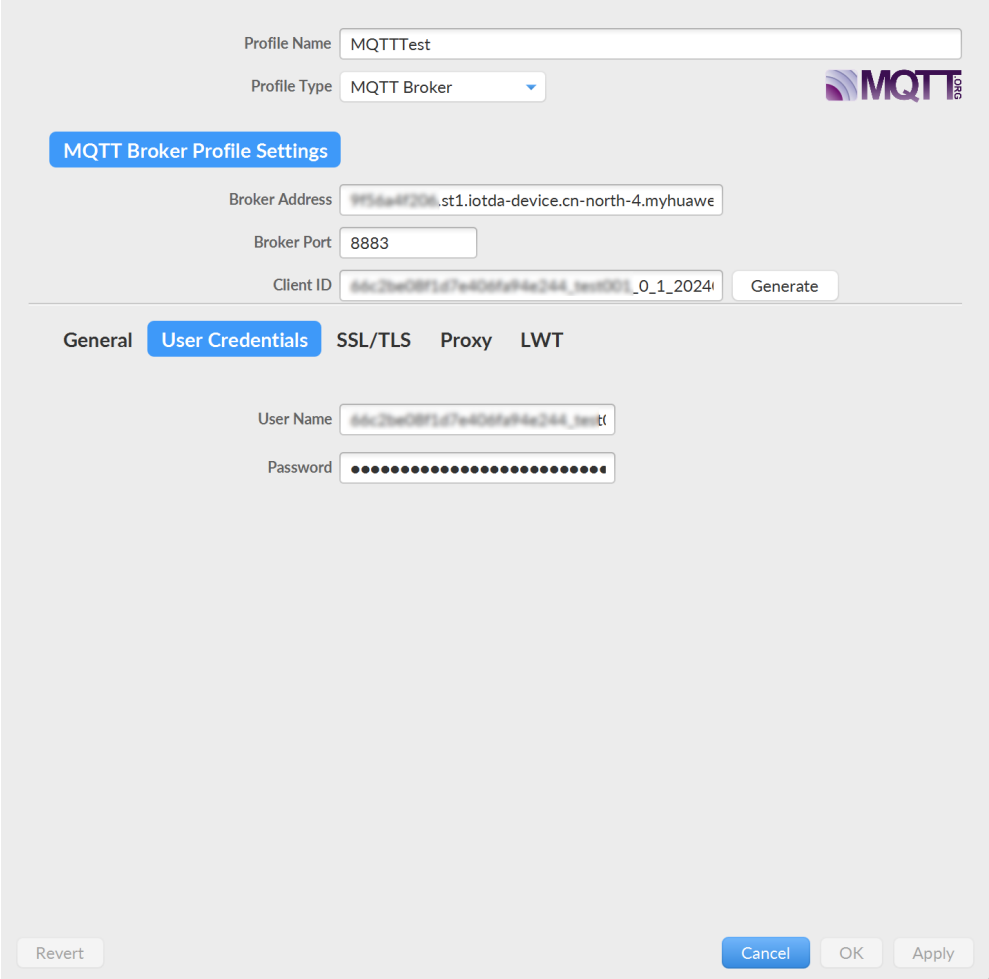


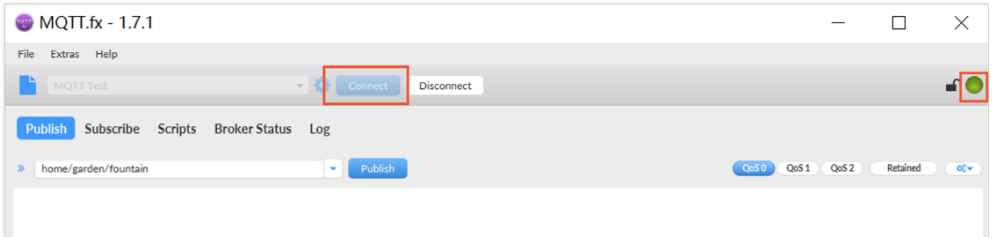
Table 1-3 MQTT.fx connection parameters

Parameter	Description	Example Value
Profile Name	Name of the configuration file.	Enter **MQTT Test**.
Profile Type	Type of the connection to be configured.	The value is fixed at **MQTT Broker**, indicating that the MQTT server is connected.
Broker Address	Access address of the MQTT server.	MQTTS access address of the device. Obtain the access address from **Overview** > **Platform Access** > **MQTTS**.

Parameter	Description	Example Value
Broker Port	Access port of the MQTT server.	Enter **8883**.
Client ID	IoTDA can send and receive messages only after device access authentication is successful.	Go to the device details page, find **MQTT Connection Parameter**, and click **View** to check the clientId, username, and password.
User Name		
Password		
SSL/TLS		-
Enable SSL/TLS	Whether to use the SSL or TLS encryption protocol.	Yes.
Protocol	Protocol version.	TLS 1.2
CA certificate file	CA certificate file.	Choose **Overview** > **Platform Access** > **MQTTS** and click **Download CA Certificate** to obtain the CA certificate file.

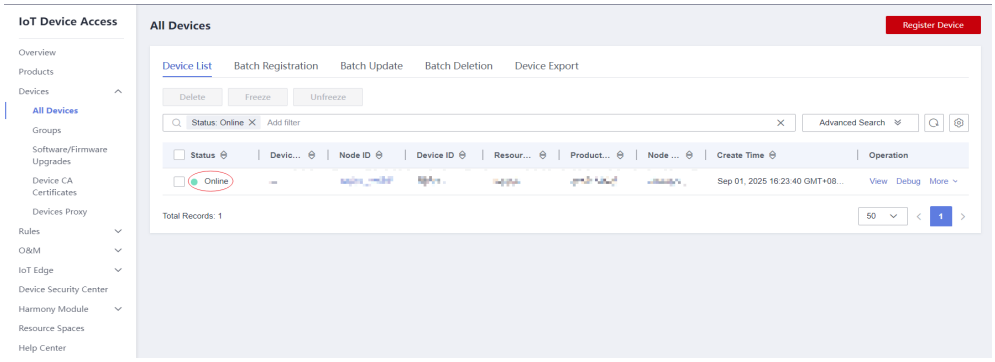
Step 3 After setting the parameters, click **Connect**. If the icon in the upper right corner turns green, MQTT.fx has been connected to IoTDA. If the icon in the upper right corner turns red, the connection fails. Click the **Log** tab to check logs, modify the configuration based on the log information, and try again.

Figure 1-5 Device connection via MQTT.fx



If the information is correct, a device connection success is displayed in the log and you can view the device status on IoTDA, as shown in the following figure.

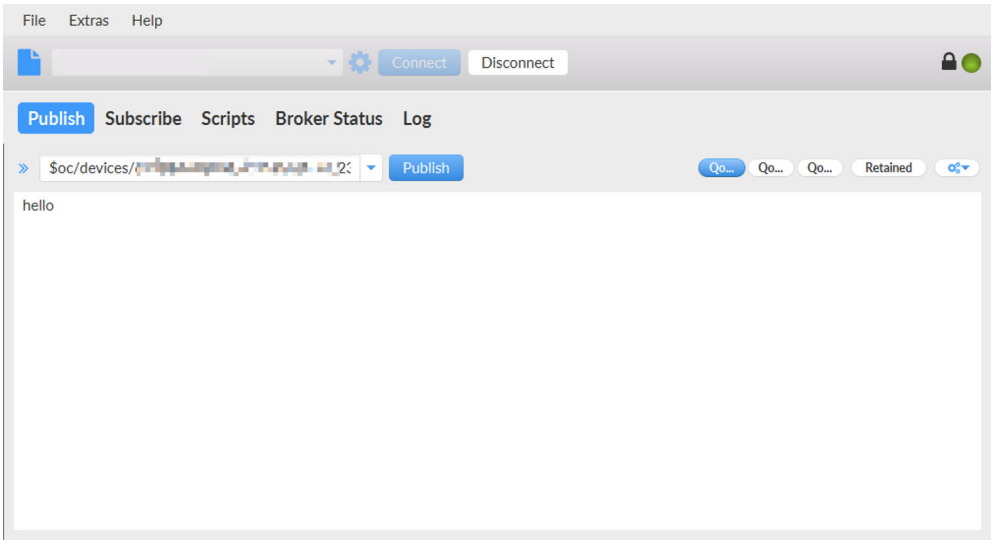
Figure 1-6 Device connection via MQTT.fx succeeded



Step 4 Use MQTT.fx to send messages.

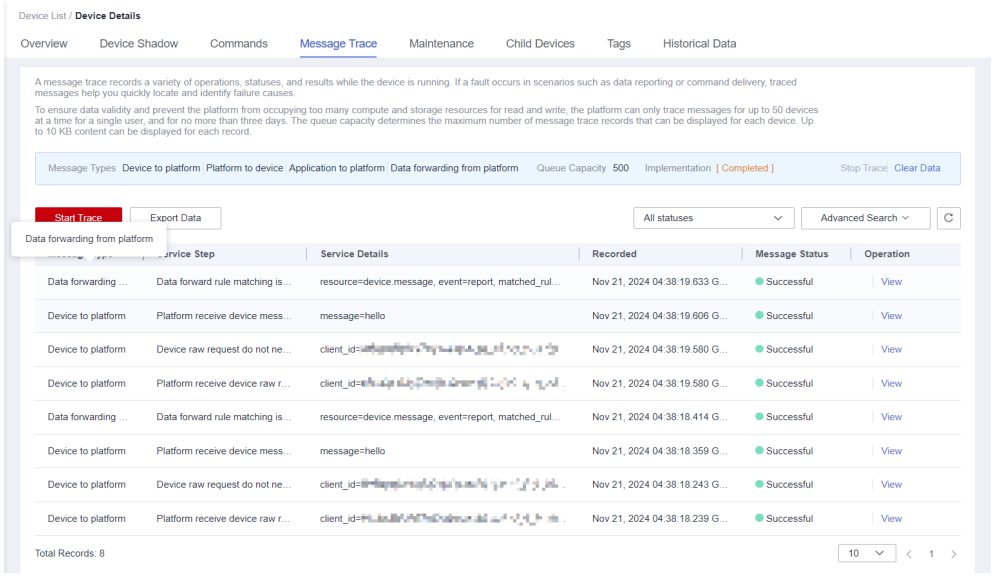
1. Click the **Publish** tab on the MQTT.fx client.
2. Enter the topic name in the **Topic** text box on the left, enter the message content in the **Message** text box, for example, **hello**, and click **Publish** on the right to send the message.

   Figure 1-7 Message sending via MQTT.fx

3. On the IoTDA console, choose **Devices** > **All Devices** in the navigation pane. On the displayed page, click **View** of the corresponding device. On the **Message Trace** page, check the messages sent by MQTT.fx.

Figure 1-8 Message viewing via MQTT.fx

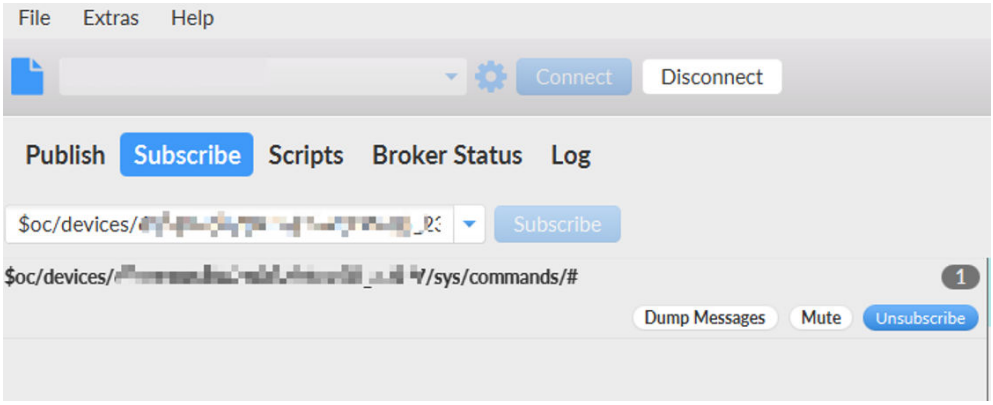


4. After MQTT.fx sends messages to the platform, configure data forwarding rules to forward the messages to message middleware, storage, data analysis, or service applications.

Step 5 Use MQTT.fx to receive messages.

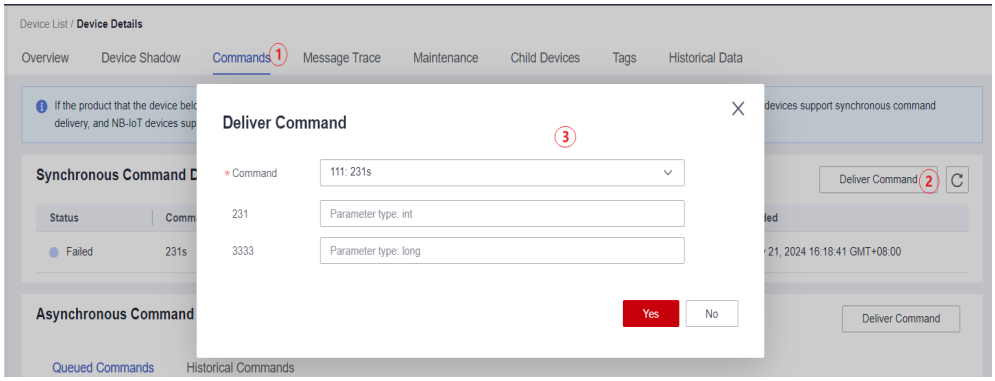
1. Click the **Subscribe** tab on the MQTT.fx client.
2. On the displayed tab page, enter the topic name in the **Topic** text box on the left and click **Subscribe**. After the topic is subscribed to, check the topic in the subscription list.

Figure 1-9 Topic subscription via MQTT.fx



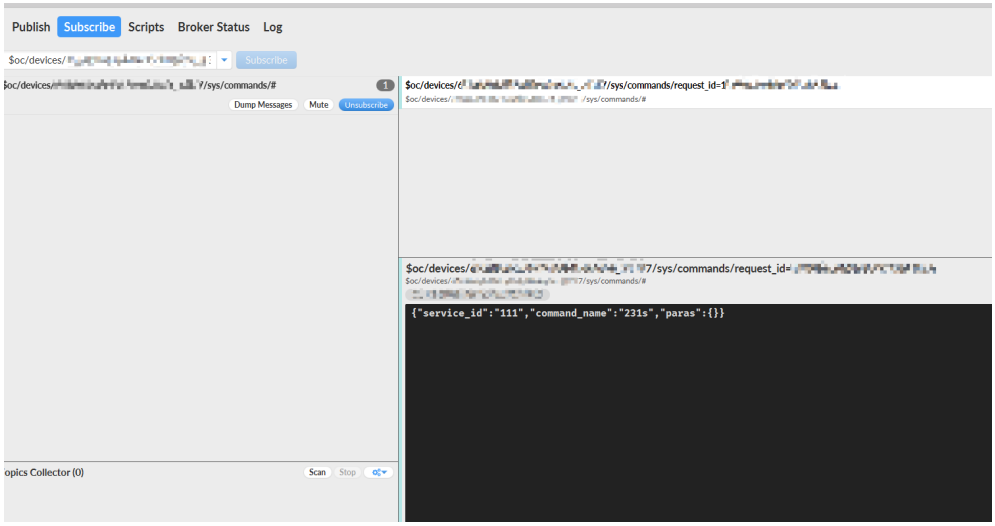
3. On the IoTDA console, choose **Devices** > **All Devices** in the navigation pane. In the device list, click a device to go to its details page.
4. Click the **Commands** tab. Click **Deliver Command** in the **Synchronous Command Delivery** area, select a command, enter the parameters to be delivered, and click **Yes**.

Figure 1-10 Command delivery



5. On the MQTT.fx client, click the **Subscribe** tab. The message received from the subscribed topic is displayed.

Figure 1-11 Message receiving



----End

1.3 Connecting NB-IoT Devices

This topic uses the NB-IoT device simulator as an example to describe how to connect devices to IoTDA using CoAP for data reporting and command delivery.

Prerequisites

You have enabled the IoTDA service through your enterprise administrator.

Creating a Product

Step 1 Log in to the IoTDA console.

Step 2 Choose **Products** in the left navigation pane.

Step 3 Click **Create Product** in the upper right corner, set the parameters based on **Table 1-1**, and click **OK**.

Table 1-4 Creating a product

Parameter	Description
Resource Space	IoTDA automatically allocates the created product to the default resource space. • You can select another resource space from the drop-down list box as you want. • You can create a resource space in **Resource Spaces** and allocate your products in it.
Product Name	Enter a name unique in the resource space.
Protocol	In this example, select **CoAP**.
Data Type	In this example, select **Binary**. You need to develop a codec on the IoTDA console to convert binary code data reported by devices into JSON data. The devices can communicate with IoTDA only after the JSON data delivered by IoTDA is parsed into binary code.
Manufacturer	Enter the manufacturer name of the device.
Device Type	Set this parameter based on service requirements.

Step 4 On the product details page, add the **service** service, add the **level** property, add the **command** command, and add the **shift** command parameter. For the command parameter, set **Data Type** to **Enumeration**. The enumerated values are **ON** and **OFF**, which are separated by commas (,).

Figure 1-12 Adding a service

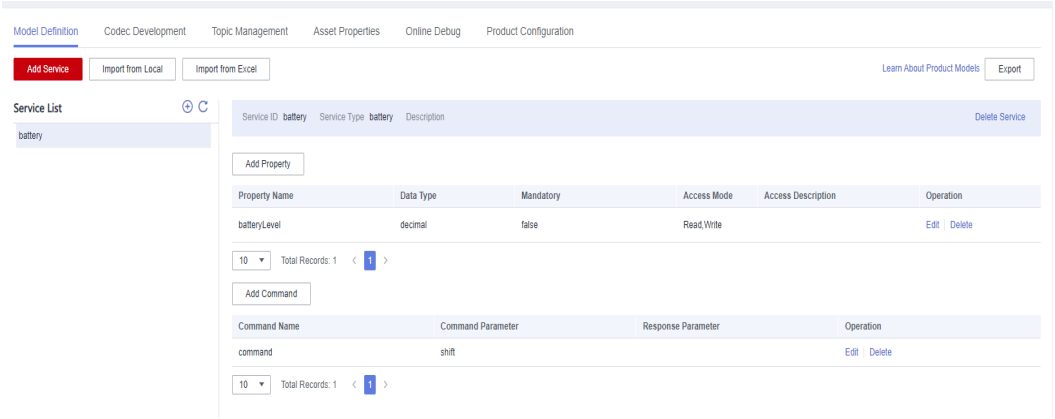
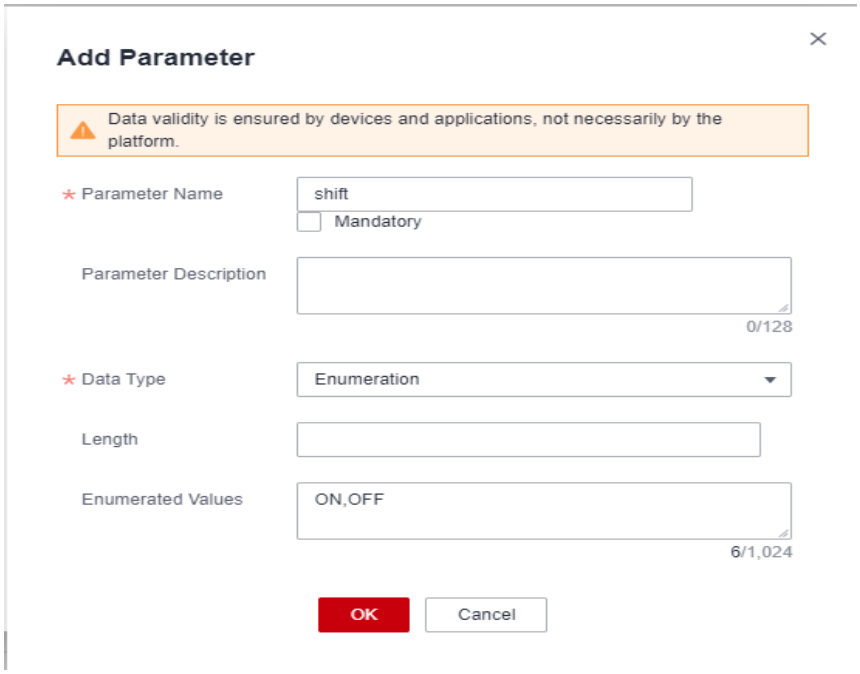
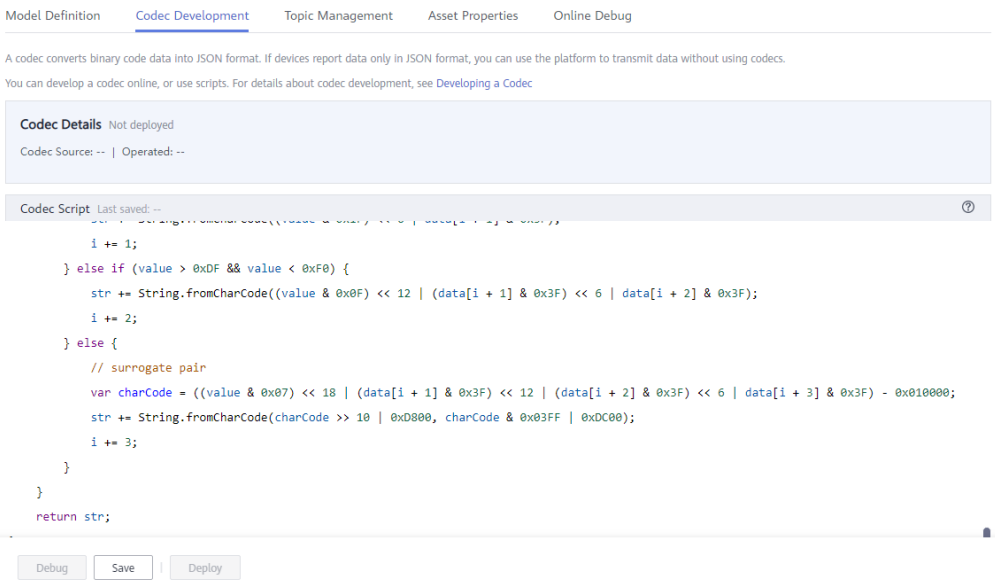


Figure 1-13 Adding a parameter



Step 5 On the **Codec Development** tab page, click the **Edit Script** tab, and copy the script file content to the script box, as shown in **Figure 1-14**.

Figure 1-14 Codec development



Copy the following script content to the script box.

```
// Data delivery types
var EVENT_DOWN='event_down';
var PROPERTY_SET='property_set';
var COMMAND ='command';
var PROPERTY_GET='property_get';
// Data reporting types
```



```
var EVENT_UP = 'event_up';
var PROPERTIES_UP='property_report';
var MESSAGE_UP='message_up';
var COMMAND_RESPONSE='command_response';
var PROPERTY_SET_RESPONSE='property_set_response';
var PROPERTY_GET_RESPONSE='property_get_response';

var MSG_TYPE_LIST = {
    0: PROPERTIES_UP,
    1: EVENT_UP,
    2: MESSAGE_UP
};

// Command delivery to NB-IoT devices
var NBCOMMAND ='commands';


// Topic customization
    // Property reporting
var SELT_DEFINE_PROPERTIES_TOPIC='myTopic/propertiesReport';
// Event reporting
var SELT_DEFINE_EVENT_TOPIC='myTopic/EventUp';
// Message reporting
var SELT_DEFINE_MESSAGE_TOPIC='myTopic/MessageUp';
// Subscription to event delivery
var SELT_DEFINE_EVENT_DOWN_TOPIC='myTopic/EventDown';
// Subscription to command delivery
var SELT_DEFINE_COMMAND='myTopic/Command';
// Subscribe to property settings
var SELT_DEFINE_PROPERTIES_SET = 'myTopic/propertiesSet';
// Subscribe to property query
var SELF_DEFINE_PROPERTIES_GET = 'myTopic/propertiesGet';


function decode(message, context) {
    var result = {};
    if (invalid(message)) {
        return JSON.stringify(result);
    }
    var flag = fromUTF8Array(message.payload);
    var uint8Array = new Uint8Array(message.payload.length);
    for (var i = 0; i < message.payload.length; i++) {
        uint8Array[i] = message.payload[i] & 0xff;
    }

    var dataView = new DataView(uint8Array.buffer, 0);
    var transType = dataView.getInt8(0);
    if(transType == 0){
        var serviceId = 'service';
        var level = dataView.getInt8(1);
        var jsonObj = {"msg_type":"properties_report","services":[{"service_id":serviceId,"properties":
{"level":level}}]};
        var type = jsonObj.msg_type
        var messageObj = {
            msg_type: type,
            message: jsonObj
        }
    }else{
        var payloadObj = JSON.parse(fromUTF8Array(message.payload));
        var messageObj = {"msg_type":"command_response",
                          "message":{"request_id":payloadObj.mid,
                          "result_code": payloadObj.errcode,"result_desc":
"success","command_name":"reboot","service_id":"service","paras":payloadObj.body}}
    }
    return JSON.stringify(messageObj);
}

// Delivery from the server
function encode(msgType, message, context) {
```

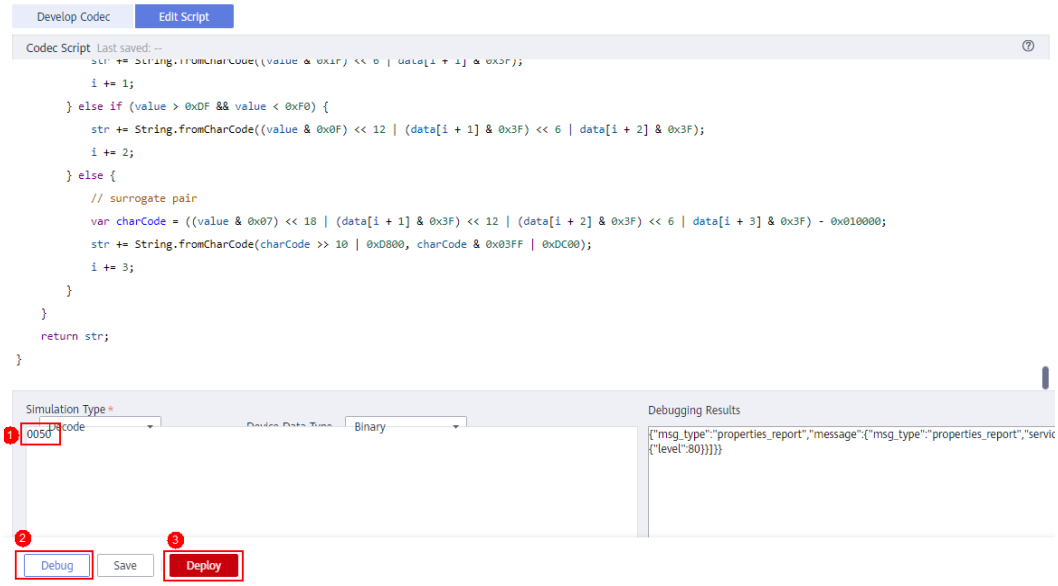
```
    var jsonObj = JSON.parse(message);
    resultMessageObj = {
        uri: "19/0/0",
        payload: toUTF8Array(jsonObj.paras.shift)
    };
    resultObj = {
        action: "commands",
        message: resultMessageObj
    };
    return JSON.stringify(resultObj);
}


function invalid(message) {
    return message === null || JSON.stringify(message) == "{}";
}
//byte ->string
function fromUTF8Array(data) { // array of bytes
    var str = '';
    for (var i = 0; i < data.length; i++) {
        var value = data[i];
        if (value < 0x80) {
            str += String.fromCharCode(value);
        } else if (value > 0xBF && value < 0xE0) {
            str += String.fromCharCode((value & 0x1F) << 6 | data[i + 1] & 0x3F);
            i += 1;
        } else if (value > 0xDF && value < 0xF0) {
            str += String.fromCharCode((value & 0x0F) << 12 | (data[i + 1] & 0x3F) << 6 | data[i + 2] & 0x3F);
            i += 2;
        } else {
            // surrogate pair
            var charCode = ((value & 0x07) << 18 | (data[i + 1] & 0x3F) << 12 | (data[i + 2] & 0x3F) << 6 |
data[i + 3] & 0x3F) - 0x010000;
            str += String.fromCharCode(charCode >> 10 | 0xD800, charCode & 0x03FF | 0xDC00);
            i += 3;
        }
    }
    return str;
}
function buffer_uint8(value) {
    var uint8Array = new Uint8Array(1);
    var dataView = new DataView(uint8Array.buffer);
    dataView.setUint8(0, value);
    return [].slice.call(uint8Array);
}
function buffer_int16(value) {
    var uint8Array = new Uint8Array(2);
    var dataView = new DataView(uint8Array.buffer);
    dataView.setInt16(0, value);
    return [].slice.call(uint8Array);
}

function toUTF8Array(str) {
    var utf8 = [];
    for (var i = 0; i < str.length; i++) {
        var charcode = str.charCodeAt(i);
        if (charcode < 0x80) {
            utf8.push(charcode);
        } else if (charcode < 0x800) {
            utf8.push(0xc0 | (charcode >> 6),
                0x80 | (charcode & 0x3f));
        } else if (charcode < 0xd800 || charcode >= 0xe000) {
            utf8.push(0xe0 | (charcode >> 12),
                0x80 | ((charcode >> 6) & 0x3f),
                0x80 | (charcode & 0x3f));
        } else {
            // surrogate pair
            i++;
            charcode = 0x10000 + (((charcode & 0x3ff) << 10)
```



```
                | (str.charCodeAt(i) & 0x3ff))
            utf8.push(0xf0 | (charcode >> 18),
                0x80 | ((charcode >> 12) & 0x3f),
                0x80 | ((charcode >> 6) & 0x3f),
                0x80 | (charcode & 0x3f));
        }
    }
    return utf8;
}
function Str2Bytes(str){
    var pos = 0;
   var len = str.length;
   if(len %2 != 0)
   { return null; }
   len /= 2;
   var hexA = new Array();
   for(var i=0; i<len; i++)
   {
      var s = str.substr(pos, 2);
      var v = parseInt(s, 16);
      hexA.push(v);
      pos += 2;
   }
   return hexA;
}
//byte ->string
function fromUTF8Array(data) { // array of bytes
    var str = '';
    for (var i = 0; i < data.length; i++) {
        var value = data[i];
        if (value < 0x80) {
            str += String.fromCharCode(value);
        } else if (value > 0xBF && value < 0xE0) {
            str += String.fromCharCode((value & 0x1F) << 6 | data[i + 1] & 0x3F);
            i += 1;
        } else if (value > 0xDF && value < 0xF0) {
            str += String.fromCharCode((value & 0x0F) << 12 | (data[i + 1] & 0x3F) << 6 | data[i + 2] & 0x3F);
            i += 2;
        } else {
            // surrogate pair
            var charCode = ((value & 0x07) << 18 | (data[i + 1] & 0x3F) << 12 | (data[i + 2] & 0x3F) << 6 |
data[i + 3] & 0x3F) - 0x010000;
            str += String.fromCharCode(charCode >> 10 | 0xD800, charCode & 0x03FF | 0xDC00);
            i += 3;
        }
    }
    return str;
}
```

Step 6 Enter **0050** in the text box in the lower left corner and click **Debug**. After the system displays a message indicating that the script debugging is successful, click **Deploy**.

Figure 1-15 Script-based development



----End

Registering a Device

Step 1 In the left navigation pane, choose **Devices** > **All Devices**, click **Register Device** in the upper right corner, set the parameters based on the table below, and click **OK**.

Table 1-5 Registering a device

Parameter	Description
Resource Space	Select the resource space that the device belongs to.
Product	Select the product that the device belongs to. You can select a product only when it is available on the **Products** page. If not, create a product first. For details, see **Creating a Product**.
Node ID	Customize a unique physical identifier for the device. For an NB-IoT device, this parameter is usually set to its IMEI or MAC address. The device ID and secret are returned after the registration, and the device uses them to connect to IoTDA.
Device Name	Enter a custom device name.
Authentication Type	Select **Secret** so that the device uses the secret for identity verification. NB-IoT devices do not support certificate authentication.
Secret	Specify a secret used for device access. If no secret is specified, IoTDA automatically generates one.

Step 2 Save the device ID and secret. They are used for authentication when the device attempts to access IoTDA.

NOTE

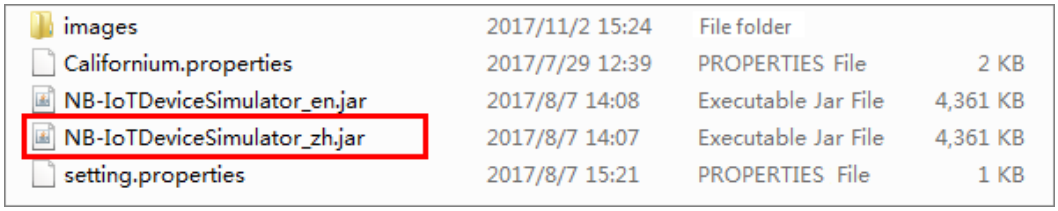
If the secret is lost, you can reset the secret. The secret generated during device registration cannot be retrieved.

----End

Performing Connection Authentication

Step 1 Download and decompress the **NB-IoT device simulator**, and double-click **NB-IoTDeviceSimulator_en.jar** to run the simulator.

Figure 1-16 NB-IoT device simulator



The package contains the following files:

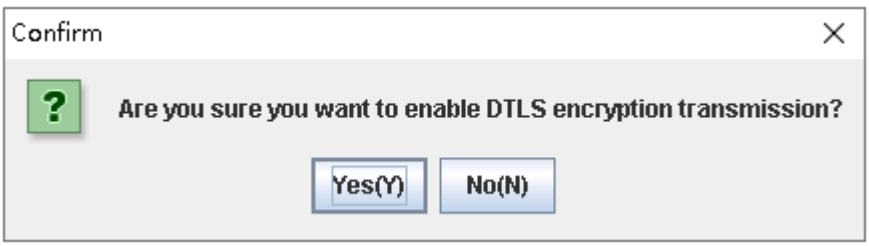
- **Californium.properties**: Simulator configuration file
- **NB-IoTDeviceSimulator_en.jar**: English simulator
- **NB-IoTDeviceSimulator_zh.jar**: Chinese simulator
- **setting.properties**: Configuration file for connecting the device simulator to IoTDA

NOTE

To run the JAR package in Windows, install JDK 1.7 or later and configure environment variables.

Step 2 When a message is displayed requesting you to confirm whether to enable DTLS encrypted transmission, click **Yes**.

Figure 1-17 Enabling DTLS transmission



Step 3 Enter **IP address** and **VerifyCode**, and click **Register Device** to bind the simulator to IoTDA.

Figure 1-18 Binding a device

Set the three parameters as follows:

- **IP Address**: Domain name or IP address of IoTDA. To obtain the domain name, log in to the IoTDA console and view the domain name on the **Overview** page. To obtain the IP address or domain name, run the **ping** *DomainName* command in the CLI.
- **VerifyCode**: Node ID set during **device registration**.
- **psk**: device secret generated by the IoT platform or entered during **device registration**.

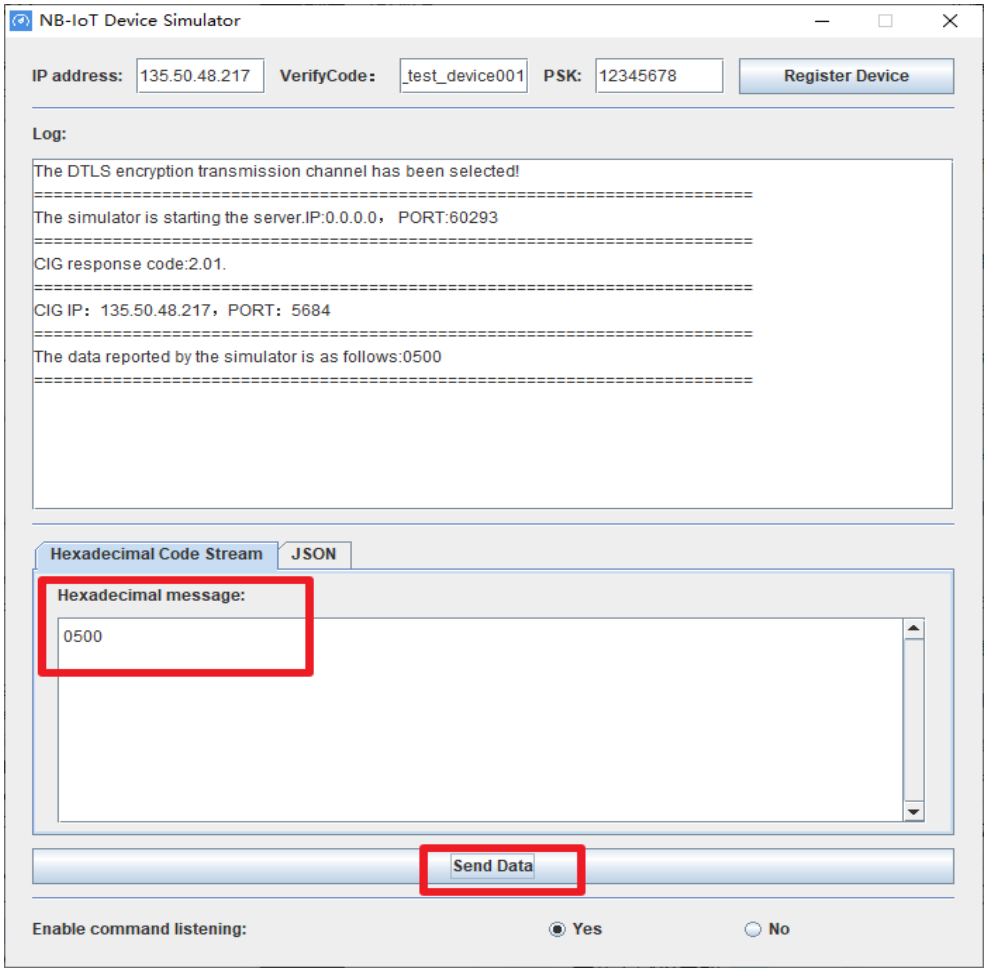
Step 4 On the IoTDA console, choose **Devices** > **All Devices** and view the device status. If the device is online, the simulator is connected to IoTDA.

----End

Reporting Data

Step 1 Simulate data reporting. If the reporting level is 80, enter a hexadecimal code stream **0050** in the NB-IoT device simulator. (**00** indicates the message type, and **50** indicates the reporting level.) Then click **Send Data**.

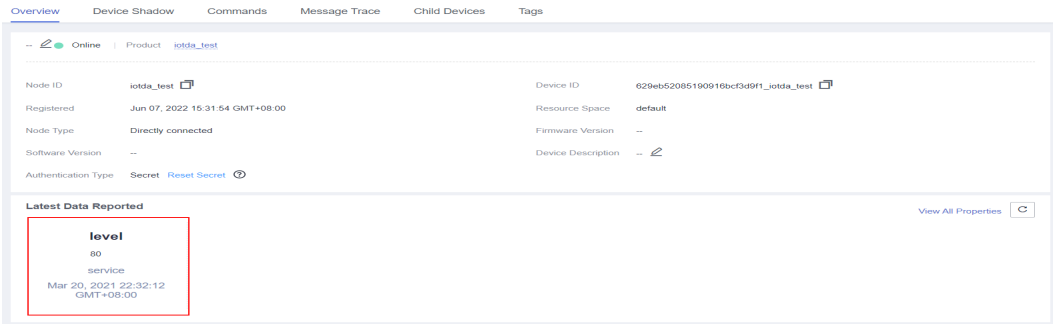
Figure 1-19 Simulating device data reporting



Step 2 On the IoTDA console, choose **Devices** > **All Devices**, and click **View** in the row that contains the device to access its details.

Step 3 On the device details page, check whether the reporting level is **80**.

Figure 1-20 Checking reported data



----End

Delivering Commands

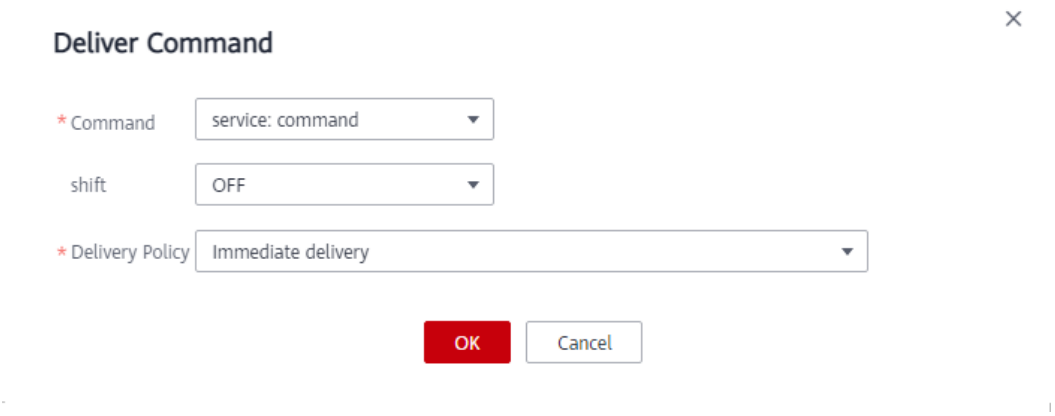
Step 1 On the device details page, click the **Commands** tab, and click **Deliver Command** in the **Asynchronous Commands** area.

Step 2 In **Application Simulator**, select **service** for **Service** and **command** for **Command**, set **Command value** to **OFF**, select **Deliver Now**, and click **OK**. See the following figure.

CAUTION

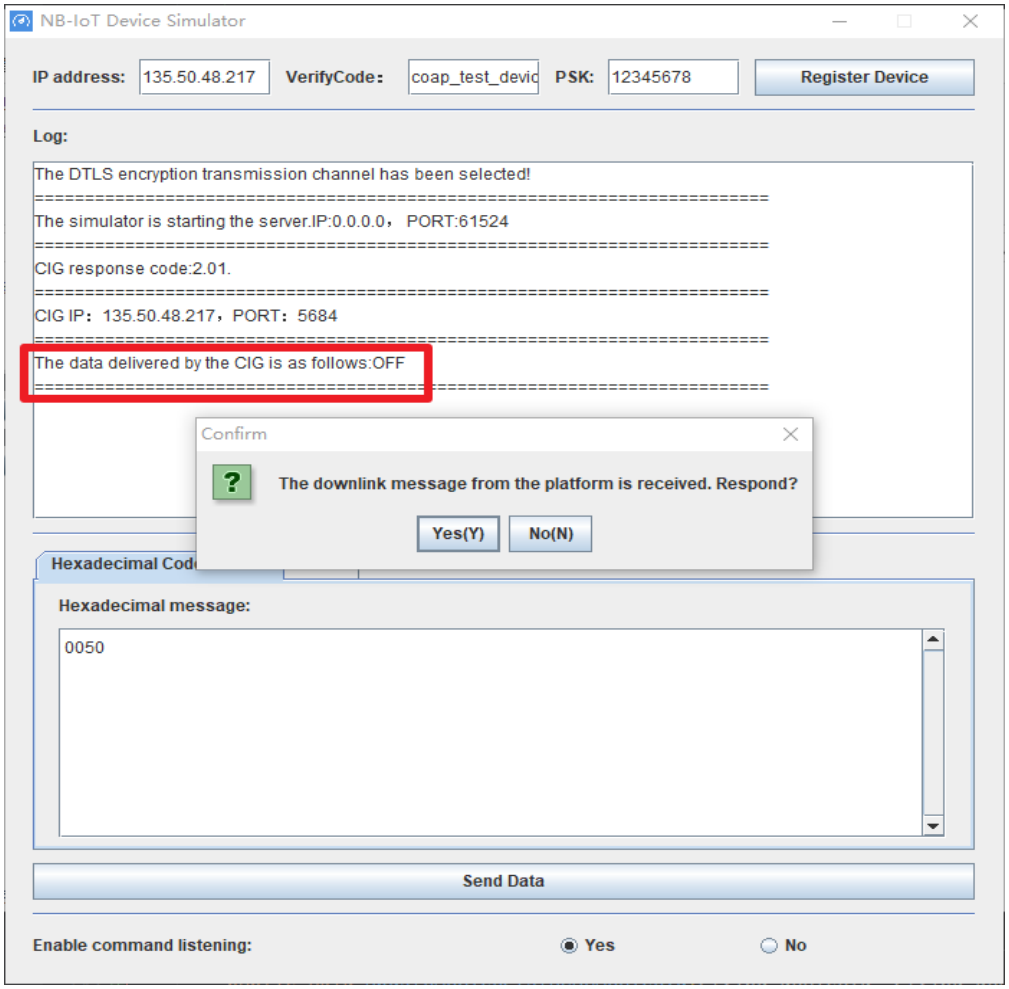
Before delivering a command, you need to add the command to the corresponding service in the product model.

Figure 1-21 Delivering Commands



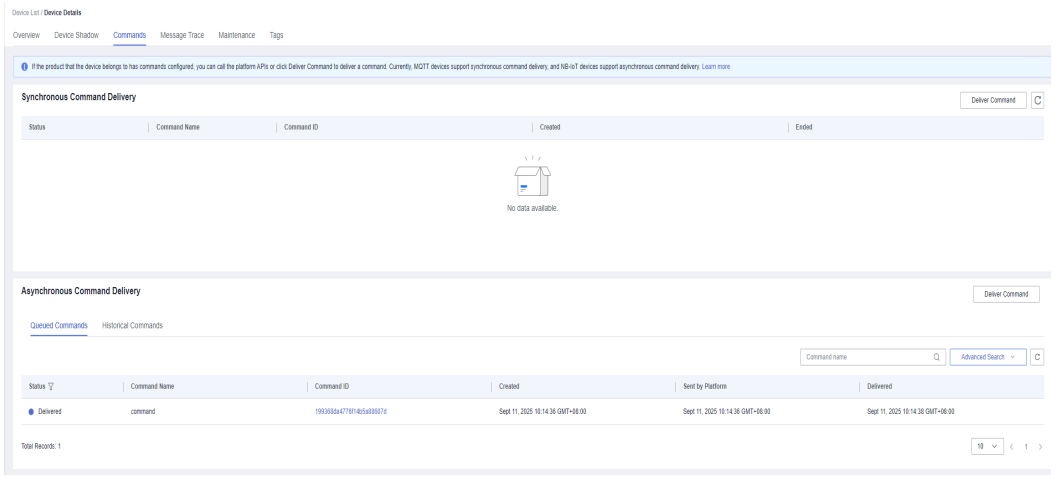
Step 3 When the simulator displays a message asking you whether to respond to the received message, click **Yes**.

Figure 1-22 Receiving downstream messages from the platform



Step 4 Check whether the command status is **Delivered** on the console.

Figure 1-23 Checking the command status



----End

2 User Guide

2.1 Products

2.1.1 Creating a Product

The first step of using IoTDA is to create a product on the console.

A product is a collection of devices with the same capabilities or features.

Procedure

Step 1 Log in to the IoTDA console.

Step 2 Choose **Products** in the left navigation pane.

Step 3 Click **Create Product** on the page, set the parameters based on the following table, and click **OK**.

Table 2-1 Creating a product

Parameter	Description
Resource Space	The platform automatically allocates the created product to the default resource space. • If you want to allocate the product to another resource space, select the required resource space from the drop-down list. • If the corresponding resource space does not exist, choose **Resource Spaces** in the left navigation pane and create a resource space.
Product Name	Name the product. The product name is unique in the resource space.

Parameter	Description
Protocol	• **MQTT** can be used by devices to access the platform. The data format can be binary or JSON. • **CoAP** can be used by NB-IoT devices with limited resources (including storage and power consumption). The data format is binary. • **HTTP** can be used by devices to access IoTDA. Currently, only message reporting and property reporting are supported. • **Modbus** can be used by devices to access IoTDA. Devices that use the Modbus protocol to connect to IoTEdge nodes are called indirectly connected devices. For differences between directly connected devices and indirectly connected devices, see **Child Devices**. • **OPC-UA** can be used by devices to access IoTDA. Devices that use the OPC UA protocol to connect to IoTEdge nodes are called indirectly connected devices. For differences between directly connected devices and indirectly connected devices, see **Child Devices**. • **OPC-DA** can be used by devices to access IoTDA. Devices that use the OPC DA protocol to connect to IoTEdge nodes are called indirectly connected devices. For differences between directly connected devices and indirectly connected devices, see **Child Devices**. • **Other** is a type of extensible proprietary protocol used for customizing product protocols.
Data Type	• **JSON**: The communication protocol between the platform and devices uses the JSON format, which is a standard format defined by the product model. • **Binary**: You can develop codecs on the console to convert the raw data reported by a device that includes binary code streams into JSON-format data that complies with the product model definition of the platform. You can also use customized topics to transparently transmit the raw data.
Manufacturer	Enter the manufacturer name of the device.
Device Type	Enter **StreetLight**, **GasMeter**, or **WaterMeter**. Set this parameter based on service requirements.
Advanced Settings	
Product ID	Set a unique identifier for the product. If this parameter is specified, the platform uses the specified product ID. If this parameter is not specified, the platform allocates a product ID.
Description	Provide a description for the product. Set this parameter based on service requirements.

Step 4 After a product is created, you can view it in the product list. You can click **Delete** to delete a product that is no longer used. After the product is deleted, its resources such as the product models will be cleared. Exercise caution when deleting a product.

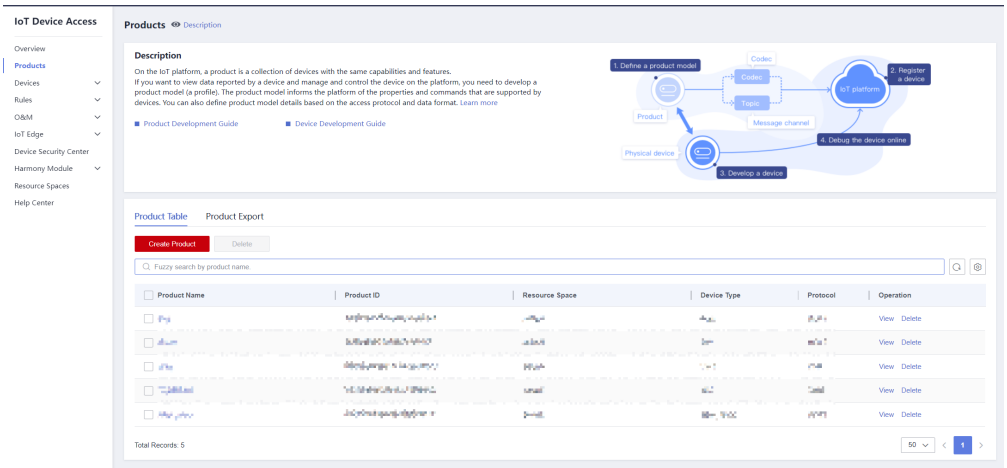
----End

2.1.2 Querying a Product

After a product is created in IoTDA, you can query the product in multiple dimensions.

Step 1 Log in to the IoTDA console and choose **Products** in the navigation pane. The product list is displayed.

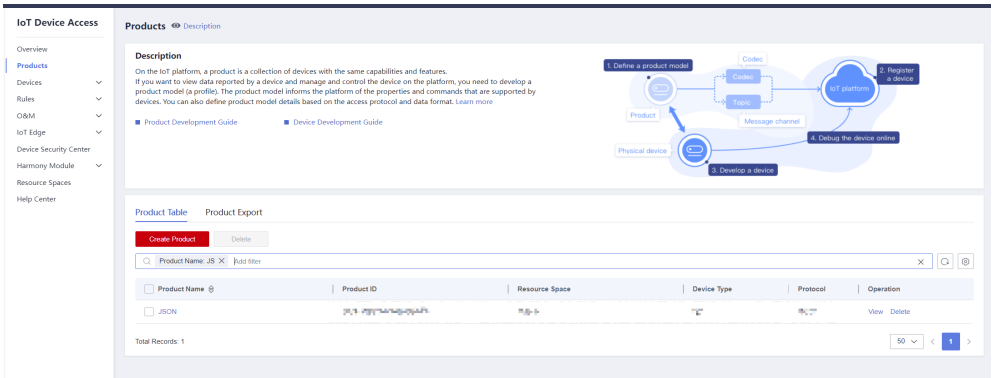
Figure 2-1 Product list



Step 2 Search for a product by product name, product ID, or device type.

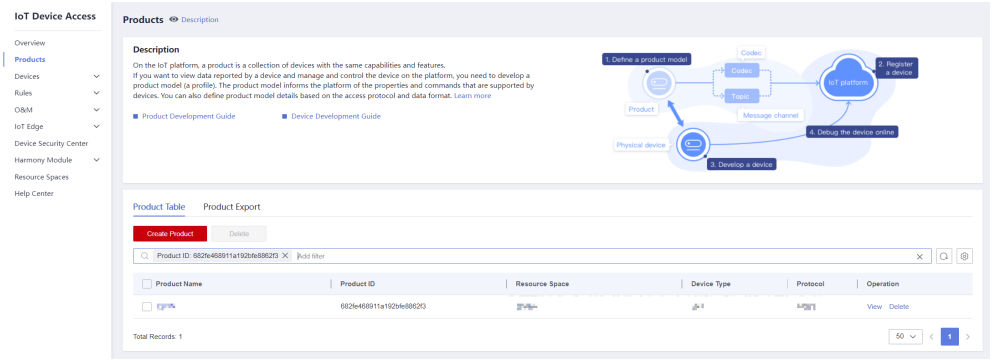
- Perform fuzzy search for products by product name. For example, select **Product Name** and search for **AB** to obtain products whose names contain "AB".

Figure 2-2 Fuzzy search



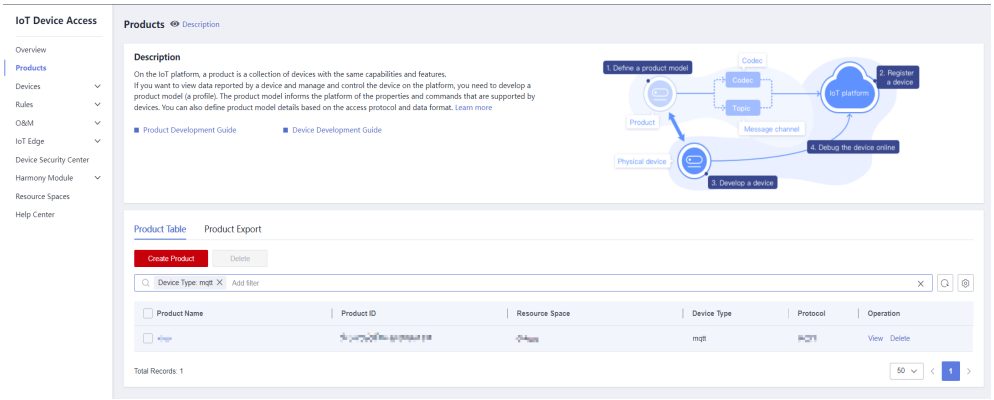
- Perform exact search for products by product ID. For example, select **Product ID** and search for **682fe468911a192bfe8862f3** to obtain the product whose ID is "682fe468911a192bfe8862f3".

Figure 2-3 Exact search by product ID



- Perform exact search for products by device type. For example, select **Device Type** and search for **MQTT** to obtain products whose device type is "MQTT".

Figure 2-4 Exact search by device type



Step 3 Delete product individually or in batches.

- Click **Delete** in the **Operation** column and delete a product as prompted.
- Select multiple products, click **Delete** above the table, and delete the selected products as prompted.

----End

More Operations

Step 1 In the product list, click **View** in the row where the product is located. On the product details page displayed, you can view basic product information, such as the product ID, product name, device type, data type, manufacturer name, resource space, and protocol type. The product ID is automatically generated by IoTDA. Other information is defined by you during product creation.

Figure 2-5 Product details

Step 2 On the product details page, you can develop product models and codecs. For details, see "Product Development" in *Developer Guide* of the corresponding service version.

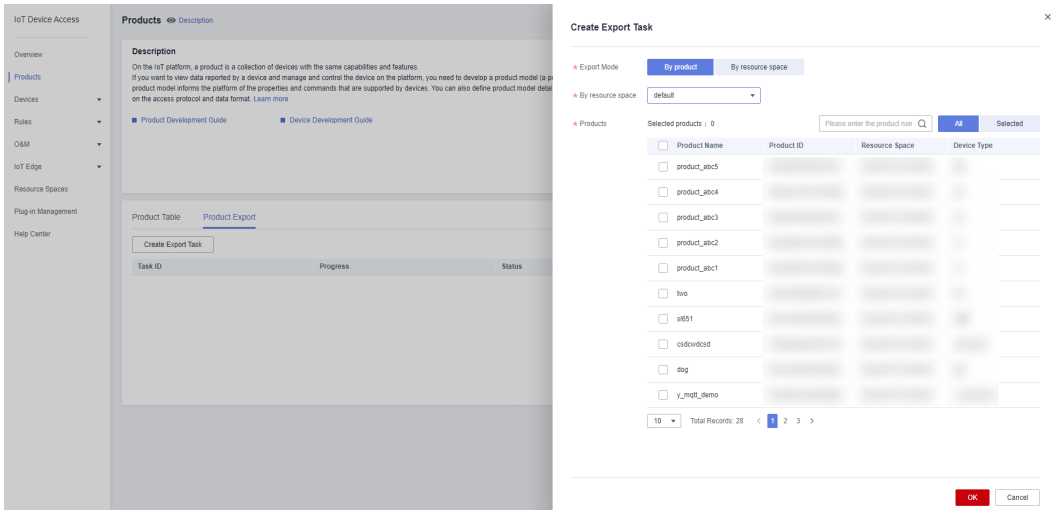
----End

2.1.3 Exporting a Product

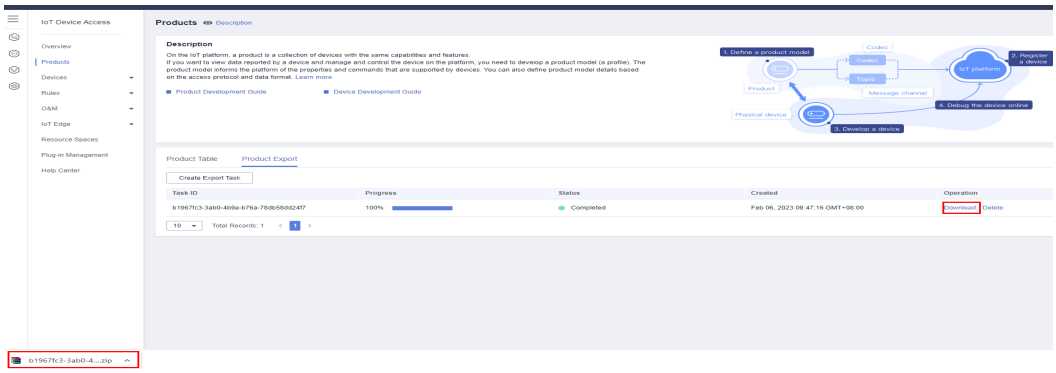
After a product is created in IoTDA, you can export products by resource space or specify products in a specific resource space to export. For details about restrictions on export tasks, see **Exporting Device Information**.

Exporting a Specified Product

Step 1 In the navigation pane, choose **Products**, click the **Product Export** tab, and click **Create Export Task**. On the page displayed, select **By product**, select a resource space, and select the product to export. Click **OK**.



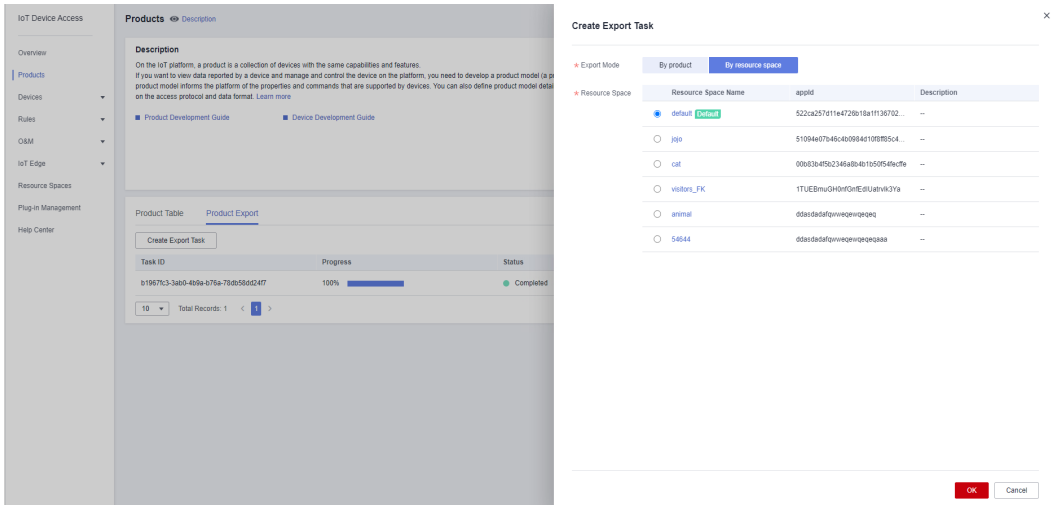
Step 2 After the export is complete, click **Download** in the export task list to obtain the product file.



----End

Exporting Products by Resource Space

Step 1 In the navigation pane, choose **Products**, click the **Product Export** tab, and click **Create Export Task**. On the page displayed, select **By resource space** and select a resource space. Click **OK**.



Step 2 After the export is complete, click **Download** in the export task list to obtain the product file.

----End

2.2 Devices

2.2.1 Registering a Device

2.2.1.1 Registering an Individual Device

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

IoTDA allows an application to call the API **Creating a Device** to register an individual device. Alternatively, you can register an individual device on the console. This topic describes the procedure on the IoTDA console.

Procedure

Step 1 Log in to the IoTDA console.

Step 2 In the left navigation pane, choose **Devices** > **All Devices**, click **Register Device** in the upper right corner, set the parameters based on **Table 2-2**, and click **Yes**.

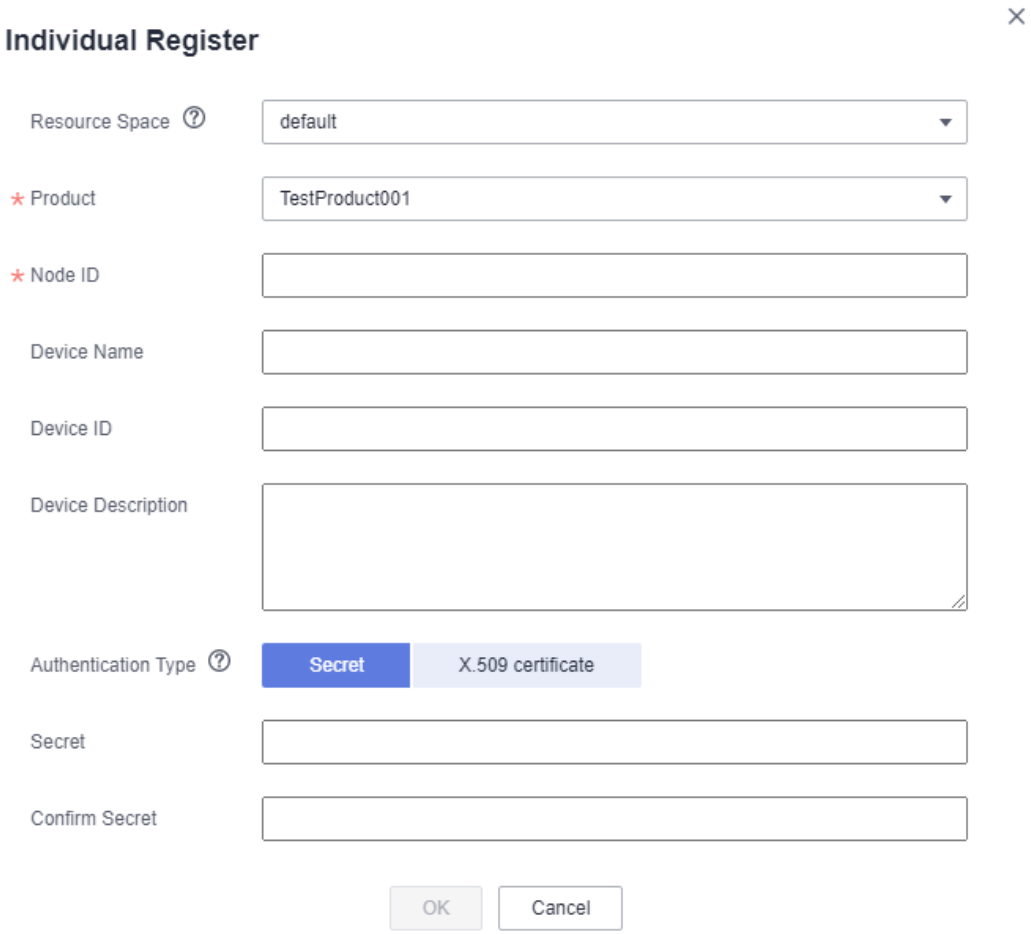
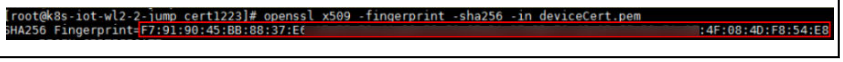


Table 2-2 Registering a device

Parameter	Description
Resource Space	Select the resource space to which the device belongs.
Product	Select the product to which the device belongs. You can select a product only when it is available on the **Products** page. If no product is available, create a product first. For details, see **Products**.
Node ID	Specify a unique physical identifier for the device, such as its IMEI or MAC address. This parameter is used by IoTDA to authenticate the device during device access. • For a native MQTT device, this parameter is customized. The device ID (corresponding to the node ID) and secret generated after the registration are used for device access. • For an NB-IoT device or a device integrated with the SDK, set the node ID to the IMEI or MAC address. The device connects to the platform using the node ID and secret entered during registration.

Parameter	Description
Device Name	Customize the name of the device.
Device ID	It uniquely identifies a device. If this parameter is carried, the platform will use the parameter value as the device ID. Otherwise, the platform will allocate a device ID, which is in the format of *[product_id]_ [node_id]*.
Device Description	Describe the device.
Authentication Type	**Secret**: The device uses the secret for identity verification. **X.509 certificate**: The device uses an X.509 certificate for identity verification. Before registering a device authenticated by an X.509 certificate, upload the device CA certificate to IoTDA and bind the device certificate to the device during device registration. For details, see **Device CA Certificates**. The server can skip the device CA certificate verification through configuration. However, this may pose high security risks. For details, see the *O&M Guide*.
Secret	This parameter is displayed when **Authentication Type** is set to **Secret**. You can customize a device secret. If you leave it blank, IoTDA automatically generates one.
Confirm Secret	Keep it the same as the entered secret.
Fingerprint	This parameter is displayed when **Authentication Type** is set to **X.509 certificate**. Enter the fingerprint of **Presetting an X.509 Certificate**. You can run the **openssl x509 -fingerprint -sha256 -in deviceCert.pem** command in the OpenSSL view to query the fingerprint. **Note: Delete the colons (:) from the obtained fingerprint when filling it.** 

Step 3 Save the device ID and secret. They are used for authentication when the device attempts to access IoTDA.

NOTE

If the secret is lost, click **Reset Secret** on the **Overview** tab page of the device to reset the secret. The secret generated during device registration cannot be retrieved.

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

----End

2.2.1.2 Registering a Batch of Devices

IoTDA allows an application to call the API **Creating a Batch Task** to register a batch of devices. Alternatively, you can perform batch registration on the console. This topic describes how to use the IoTDA console to register a batch of devices.

Procedure

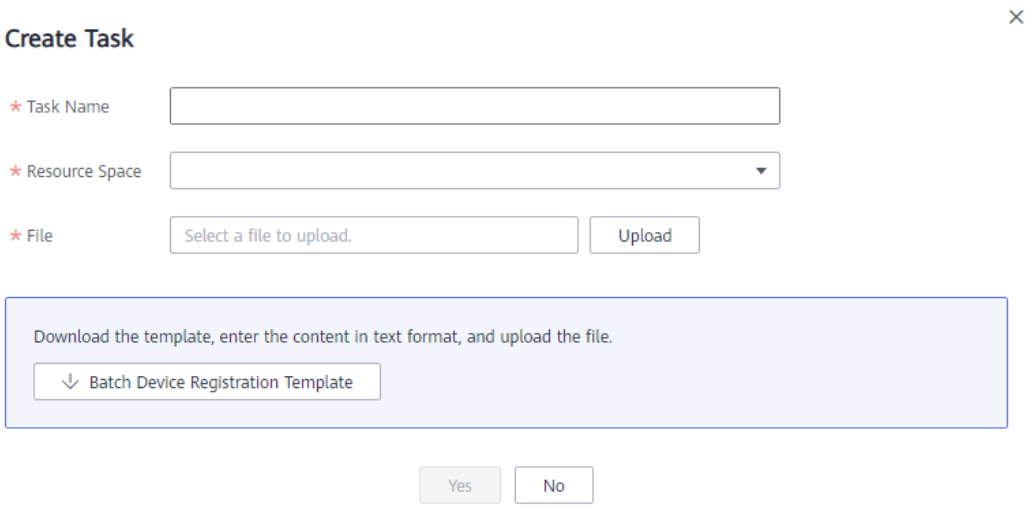
Step 1 Log in to the IoTDA console.

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

Step 3 Download the batch device registration template and fill in it. On the **Create Task** dialog box, enter the task name, select the resource space, upload the template file, and click **Yes**.

NOTE

- Enter a device secret in the template. If you leave it blank, the platform automatically generates one. If the secret is lost, you can reset the secret. The secret generated during device registration cannot be retrieved.
- A batch registration file to be uploaded can contain up to 30,000 lines.
- Batch device registration tasks are automatically deleted from the task list after 30 days.



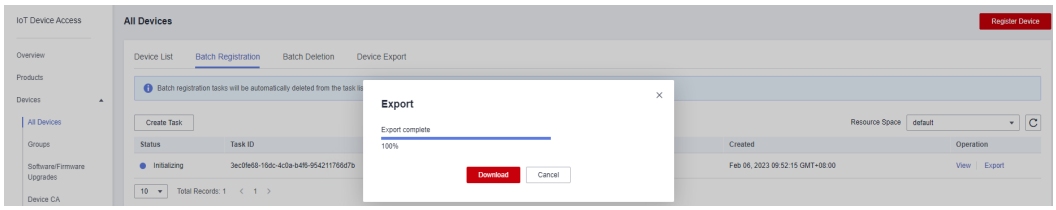
----End

2.2.1.3 Exporting Batch Device Registration/Deletion Details

You can export batch device registration or deletion details to an Excel file. The following describes how to export batch device registration details. For details about restrictions on export tasks, see **Exporting Device Information**.

Step 1 Log in to the IoTDA console.

Step 2 In the left navigation pane, choose **Devices** > **All Devices**, click the **Batch Registration** tab to view batch registration tasks. Click **Export**. Then, download and view batch device registration details.



----End

2.2.2 Managing a Device

2.2.2.1 Viewing and Deleting a Device

After a device is registered on IoTDA, you can manage the device, view device information, and freeze the device on the console.

In the navigation pane of the console, choose **Devices** > **All Devices**. By default, all devices under your account are displayed in the device list.

- You can click the text box above the list to search for devices by resource space, product name, device status, node type, registration time, device name, node ID, and device ID. Device name and node ID support fuzzy search by prefix. For example, if there are devices prefixed with **abc**, you can enter **abc** to search for the devices. Currently, device names and node IDs can contain only digits and letters. Device ID only supports exact search.
- You can also click **Advanced Search** in the upper right corner of the device list to search for devices by exact tag or asset attribute.
- You can view the **device statuses**, device name, node ID, and **device details**.
- You can click **Delete** in the row of a device to delete the device, or you can **delete devices in batches**.

  NOTE

  After a device is deleted, the related device data is deleted. Exercise caution when performing this operation.

Device Statuses

You can view the device status (online, offline, inactive, or frozen) on the IoTDA console. The table below describes the device statuses.

Status	Short-Connection Device (Such as NB-IoT Devices)	Persistent Connection Device (MQTT Device)
Online	If the device has been registered and gone online within 49 hours, the device status is **Online**.	The device is always connected to the platform.

Status	Short-Connection Device (Such as NB-IoT Devices)	Persistent Connection Device (MQTT Device)
Offline	If the device has not been registered and gone online for more than 49 hours after connecting to the platform, the platform sets the device status to **Offline**.	After the device is disconnected from the platform for 1 minute (the data is automatically updated every minute), the device status is set to **Offline**. If you manually refresh the status on the page, the device status is displayed as **Offline**.
Inactive	The device is registered with but does not connect to the platform. Initialize the device to connect it to the platform.	The device is registered with but does not connect to the platform. Initialize the device to connect it to the platform.
Frozen	The device is frozen and cannot be controlled.	The device is frozen and cannot be controlled.

Device Details

In the device list, click **View** in the row of a device to access its details.

Parameter	Description
Overview	• Viewing device information: You can view basic device information, including the node ID, device ID, node type, software version, and firmware version. – Node ID is a unique physical identifier for the device, such as its IMEI or MAC address. This parameter is used by IoTDA to authenticate the device during device access. – Device ID uniquely identifies a device. It is allocated by IoTDA during device registration and used for device access authentication and message transmission. • Resetting a secret: The secret is used for authentication when MQTT devices, NB-IoT devices, or SDK-integrated devices access IoTDA. After the secret is reset, the new secret must be updated on the device, and the device must carry the new secret for authentication during platform connection. • Viewing the latest reported data: View the latest data reported by the device to the platform.
Device Shadow	The platform provides the device shadow to cache the device status. After the device goes online, the platform delivers the cached property setting command to the device. The device can also proactively obtain the shadow data of the device from the platform. For details, see **Device Shadow**.

Parameter	Description
Commands	You can create a command delivery task for an individual device on the console. For details, see **Command Delivery**.
Message Trace	You can use message trace to record operation information, status, and results during device running. When a fault occurs in service scenarios such as data reporting and command response, message trace can help you quickly locate the fault and analyze the cause. For details, see **Message Trace**.
Child Devices	Devices can be directly or indirectly connected to IoTDA. Indirectly connected devices access IoTDA through gateways. For details, see **Child Devices**.
Tag	You can define tags and bind tags to devices. For details, see **Groups**.
Asset Properties	You can set asset properties on the device only after adding asset properties to the product. For details, see **Asset Properties**.
Device Monitoring	• IoTDA provides the device security center, which provides security detection and offline analysis. For details, see **Device Security Center**.

Deleting a Batch of Devices

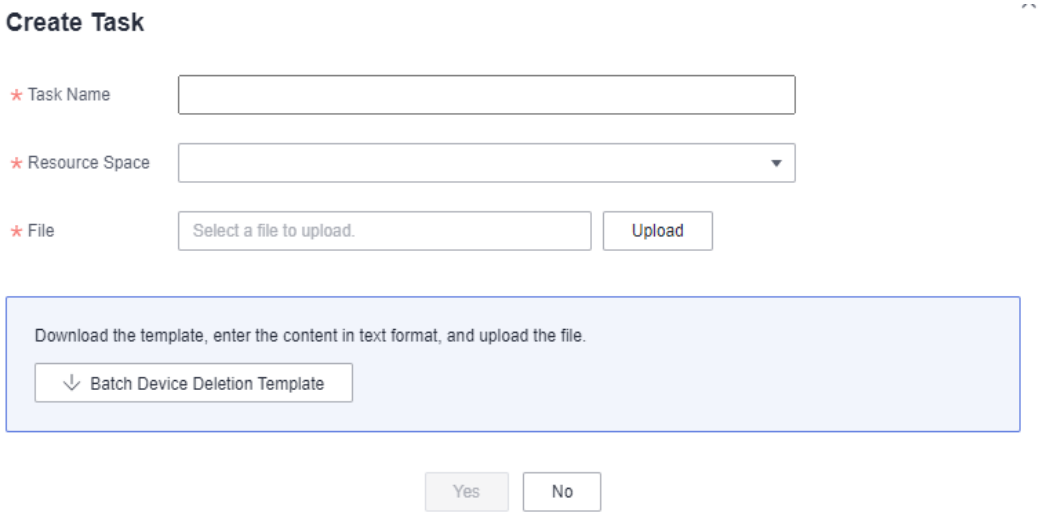
NOTE

- A batch deletion file to be uploaded can contain up to 30,000 lines.
- Batch device deletion tasks are automatically deleted from the task list after 30 days.

To delete devices in batches on the console, perform the following steps:

Step 1 Log in to the IoTDA console.

Step 2 In the left navigation pane, choose **Devices** > **All Devices**, click the **Batch Deletion** tab, and then click **Create Task**.



Step 3 Specify the parameters, download the batch device deletion template, enter the device IDs to delete in the template, and upload and submit the template file.

The task execution status and result are displayed. If the success rate is not 100%, click the task name to open the task details page and view the failure cause.

----End

2.2.2.2 Exporting Device Information

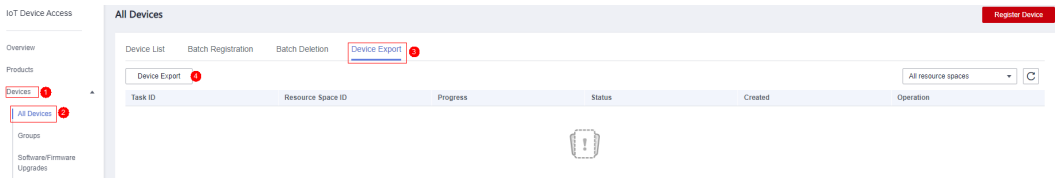
IoTDA allows users to export information of all created devices to XLS files.

NOTE

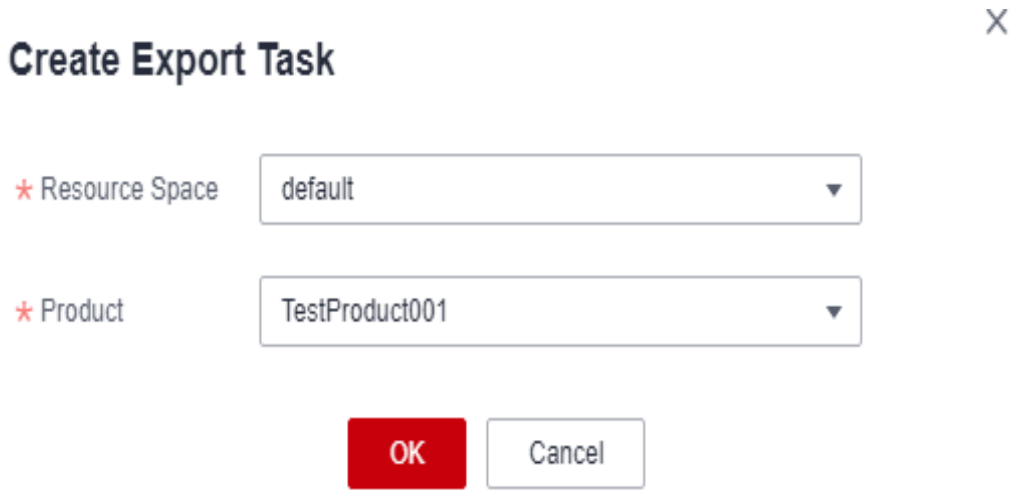
- The system allows a maximum of five concurrent export tasks because they consume many resources.
- A maximum of five export tasks can be in the **Waiting** state at a time. A successfully executed export task will be automatically deleted in one day. An export task that has been created for two days will be deleted regardless of whether it is successfully executed.
- A maximum of 100 export tasks can be created. If the limit is reached, you must delete existing export tasks before creating new ones.
- If a task is in the **Waiting** state, you cannot create another export task that has the same configuration as the current task.
- All types of export tasks are restricted by the preceding rules. For example, the total number of tasks in the **Waiting** state cannot exceed 5, regardless of whether the tasks are for product export, device export, or software/firmware upgrade.

Creating an Export Task

Step 1 In the navigation pane, choose **Devices** > **All Devices** > **Device Export**.



Step 2 Click **Device Export**, select the resource space ID and product ID of the device list to be exported, and click **OK**.

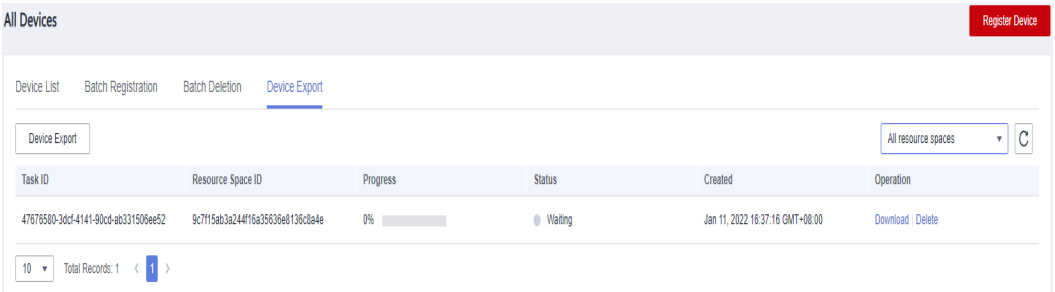


----End

Querying an Export Task

Step 1 In the navigation pane, choose **Devices** > **All Devices** > **Device Export**. Created export tasks will be displayed on the page.

Step 2 Select a resource space from the drop-down list in the upper right corner. Then, the export tasks under the selected resource space are displayed. After an export task is created, click the refresh button. The export task status is displayed as completed.



Step 3 After an export task is complete, you can click **Download** to export the file to the local PC or click **Delete** to delete the export task.

----End

2.2.2.3 Device Authentication

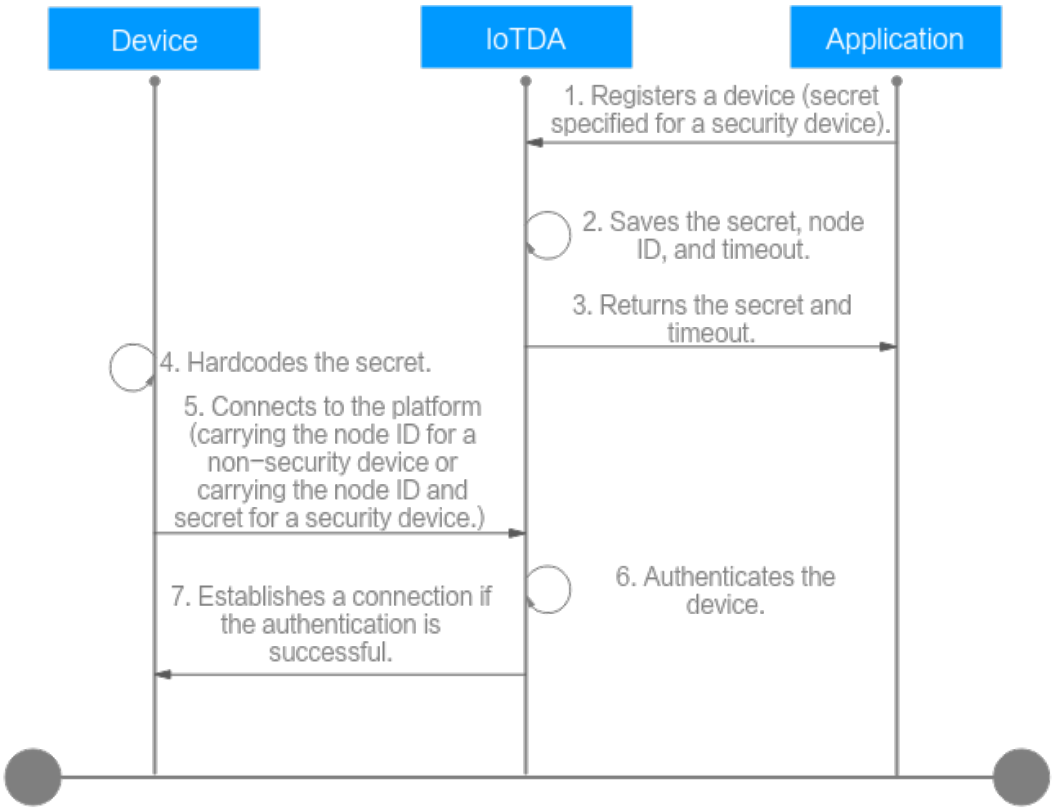
Overview

IoTDA authenticates a device when the device attempts to access the platform. The authentication process depends on the access method.

Access Type	Authentication Mode
Device using CoAP	A device is registered, either by using an application to call the API **Creating a Device** or using the console. If the device is a non-security device, it uses the node ID to get authenticated and connect to IoTDA, as shown in **Figure 2-6**.
Device using native MQTT or MQTTS	Secret for authentication A device is registered, either by using an application to call the API **Creating a Device** or using the console. The device ID and secret returned by IoTDA are also hardcoded into the device. A CA certificate is preset on MQTTS devices, but not MQTT devices. The device uses the device ID and secret to get authenticated and connect to IoTDA, as shown in **Figure 2-7**.

Authentication for Devices Using CoAP

Figure 2-6 Authentication for devices using CoAP



1. An application calls the API **Creating a Device** to register a device on IoTDA. Alternatively, you can use the console to register a device.

2–3. IoTDA allocates a secret to the device.

NOTE

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

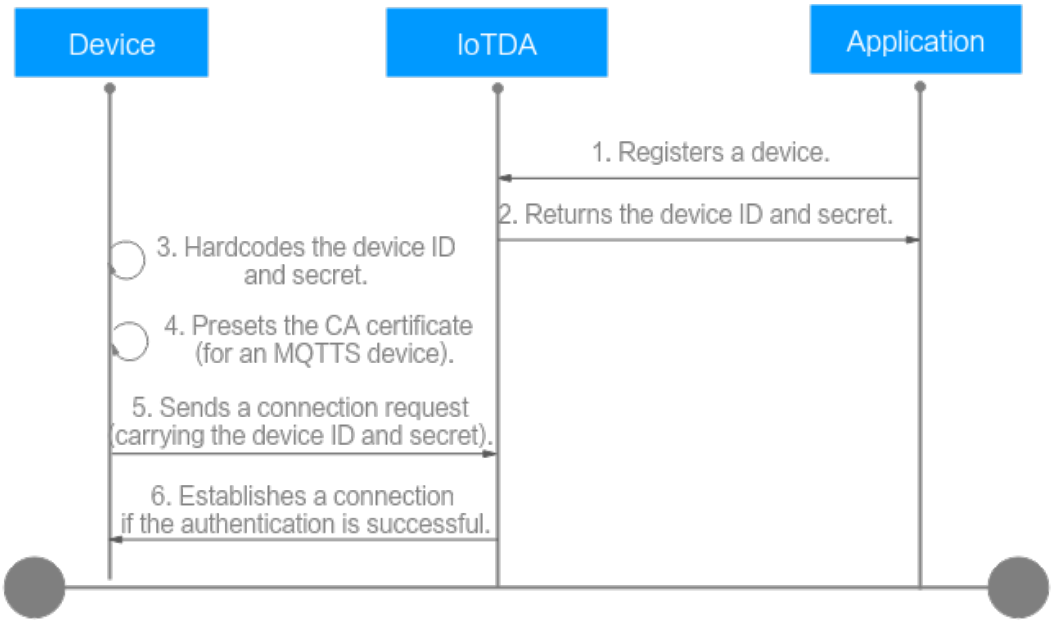
4. The user hardcodes the secret into the device hardware, software, or firmware.

5. After being powered on, the device sends a connection request carrying the node ID (such as the IMEI) and secret if it is a security device, or carrying the node ID if it is a non-security device.

6–7. If the authentication is successful, IoTDA returns a success message, and the device is connected to the platform.

Authentication for Devices Using Native MQTT or MQTTS

Figure 2-7 Authentication for devices using native MQTT or MQTTS



1. An application calls the API **Creating a Device** to register a device on IoTDA. Alternatively, you can use the console to register a device.

   NOTE

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

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

   NOTE

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

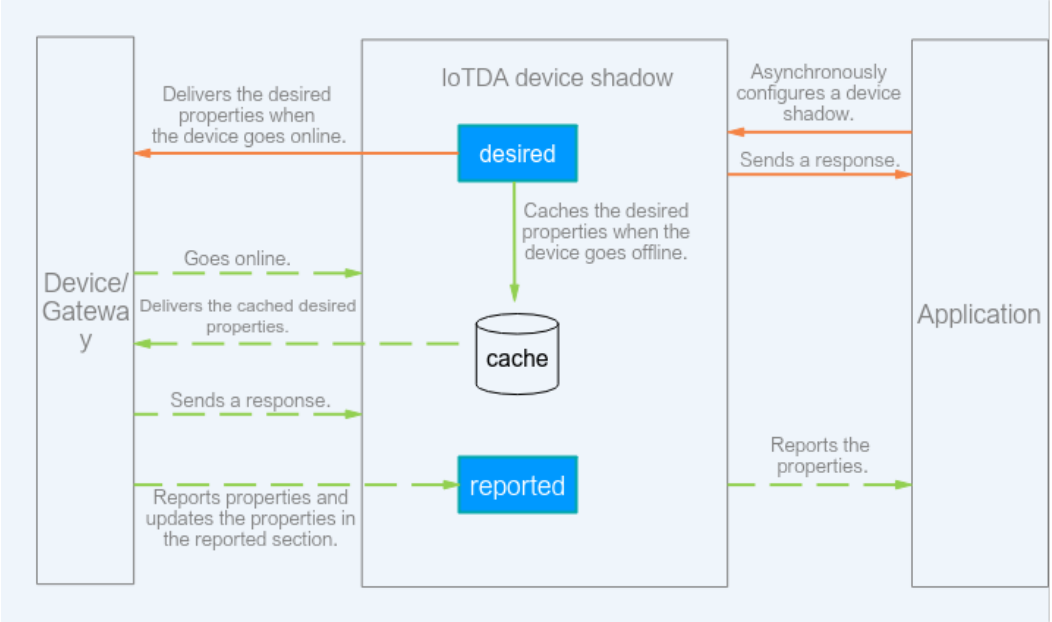
3. The user hardcodes the device ID and secret to the device hardware, software, or firmware.
4. (Optional) Preset the CA certificate on the device. This step is required only for devices connected using MQTTS.

5. After being powered on, the device sends a connection request carrying the device ID and secret.
6. If the authentication is successful, IoTDA returns a success message, and the device is connected to the platform.

2.2.2.4 Device Shadow

Overview

IoTDA allows you to create device shadows. A device shadow is a JSON structure that stores the latest device properties reported and device configuration to deliver. Each device has only one shadow. A device can retrieve and set its shadow to synchronize properties, either from the shadow to the device or from the device to the shadow.



The device shadow includes **Desired Value** and **Reported Value**.

- The desired value is used to store the configuration of device properties. To modify the service properties of a device, you can modify the desired value of the device shadow. When the device goes online, the device needs to subscribe to the topic of platform property settings first, and then the device properties will be synchronized to the device.
- The reported value is used to store the latest device properties reported by the device. When the device reports data, IoTDA changes the properties in the reported section to those reported by the device.

NOTE

- You can configure the device shadow by calling the application API or by clicking **Configure Property** on the **Device Shadow** tab page of the device details page on the IoTDA console. (The device shadow is mainly used to configure device properties. Its configuration depends on the product model.)
- The device shadow configuration is an asynchronous command. IoTDA directly returns a configuration response. Then, the platform determines whether to deliver the configuration immediately or cache the configuration based on the device status.
- When the device goes online, the device shadow delivers the desired properties to the device. After the device reports its properties, the device shadow checks whether the reported properties match the delivered ones. If they match, the shadow data is configured on the device and the cache is cleared. If they do not match, the shadow data fails to be configured on the device. When the device goes online or reports properties next time, the platform delivers the desired properties to the device again until the configuration delivery is successful.

Application Scenarios

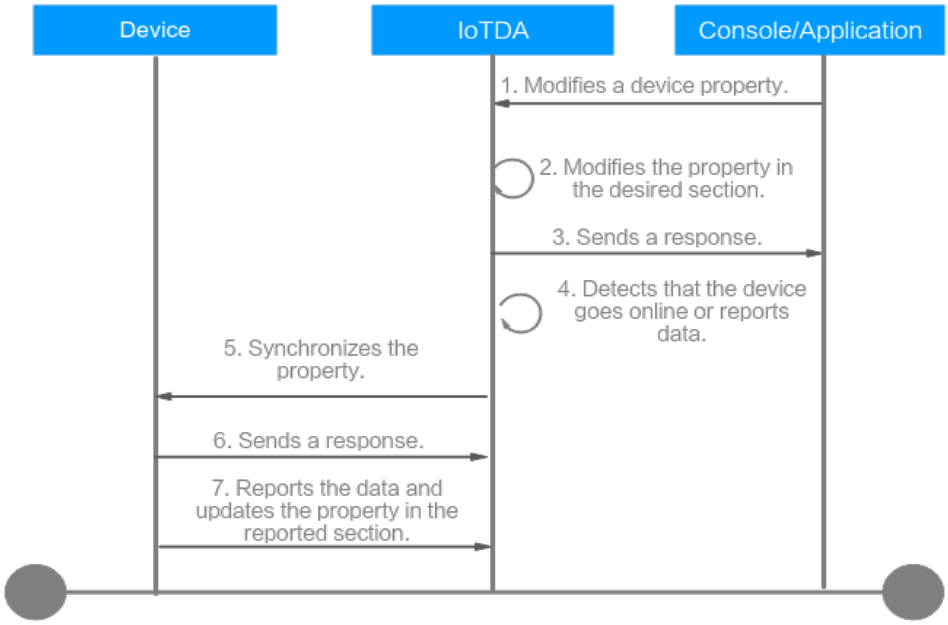
The device shadow is applicable to devices with limited resources and low power consumption or devices in the dormant state for a long time.

- Querying the latest data reported by the device and the latest online status of the device:
  - You may not be able to query the console for the mostly recent data because the device is offline or the network is unstable. With the device shadow, the platform can obtain the data from the shadow.
  - There may be too many applications simultaneously querying the device. IoT devices typically have limited processing capabilities, so too many queries can adversely affect their performance. With the device shadow, the device can synchronize its status to the shadow just once. The applications can obtain the device status from the device shadow, without reaching the real device.
- Modifying device properties: You can modify device properties on the device details page. Because the device may be offline for a long time, the modified device properties cannot be delivered to the device in time. In this case, IoTDA stores the modified device properties in the device shadow. When the device goes online, the device needs to subscribe to the topic of platform property settings first, and then the device properties will be synchronized to the device.

Service Process

Modifying a Device Property

For modifying the desired property (desired value) of the device shadow, when the device goes online, the device needs to subscribe to the topic of platform property settings first, and then the device properties will be synchronized to the device.



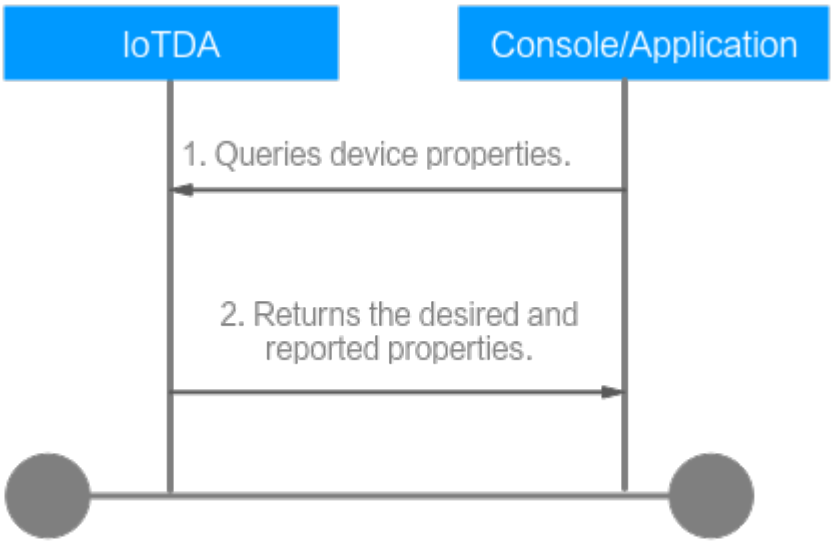
1. A user modifies a device property on the console or application. The following is an example message. For details, see "Device Shadow" in the *API Reference*.

```
PUT https://{Endpoint}/v5/iot/{project_id}/devices/{device_id}/shadow
Content-Type: application/json
X-Auth-Token: ********
Instance-Id: ********

{
  "shadow" : [ {
    "desired" : {
      "temperature" : "60"
    },
    "service_id" : "WaterMeter",
    "version" : 1
  } ]
}
```

2. IoTDA modifies the desired property.
3. IoTDA returns a response.
4. IoTDA determines that the device goes online or reports data.
5. IoTDA synchronizes device properties to the device.
6. The device returns a response.
7. When the device reports data, IoTDA changes the properties in the reported section to those reported by the device.

Querying Device Properties

The device shadow saves the most recent device properties. Once the device properties change, the device synchronizes the changes to the device shadow. Using the device shadow, a user can obtain the device status quickly regardless of whether the device is online.



1. A user queries device properties on the console or application. Example message:

```
GET https://{Endpoint}/v5/iot/{project_id}/devices/{device_id}/shadow
Content-Type: application/json
X-Auth-Token: ********
Instance-Id: ********
```

2. IoTDA returns the desired and reported properties. Example message:

   Status Code: 200 OK

```
Content-Type: application/json

{
  "device_id" : "40fe3542-f4cc-4b6a-98c3-61a49ba1acd4",
  "shadow" : [ {
    "desired" : {
      "properties" : {
        "temperature" : "60"
      },
      "event_time" : "20151212T121212Z"
    },
    "service_id" : "WaterMeter",
    "reported" : {
      "properties" : {
        "temperature" : "60"
      },
      "event_time" : "20151212T121212Z"
    },
    "version" : 1
  } ]
}
```

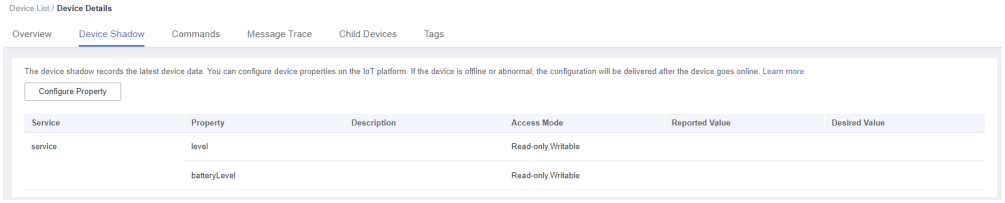
Querying and Modifying a Device Shadow

Querying a Device Shadow

Method 1: Call the API **Querying a Device Shadow**.

Method 2: Log in to the console. In the left navigation pane, choose **Devices** > **All Devices**. Click **View** in the row of a device to access its details. On the **Device Shadow** tab page, you can view the device properties, including reported and desired values.

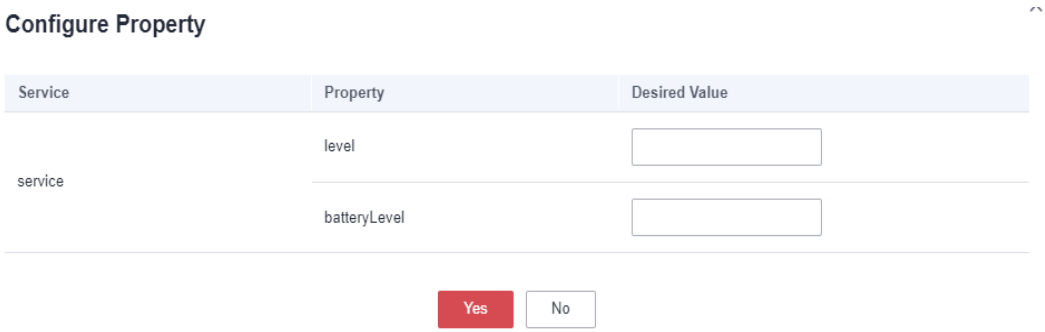
- If the reported value is inconsistent with the desired value, the desired value is highlighted. This may occur when the device is offline and the value is still in the device shadow waiting to be synchronized to the device.
- If the reported value matches the desired value, the desired value is not highlighted. The latest property reported by the device matches the desired property.



Modifying a Device Shadow

Method 1: Call the API **Configuring Desired Device Shadow Data**.

Method 2: Log in to the console. In the left navigation pane, choose **Devices** > **All Devices**. Click **View** in the row of a device to access its details. On the **Device Shadow** tab page, click **Configure Property** and enter the desired values of the service properties.

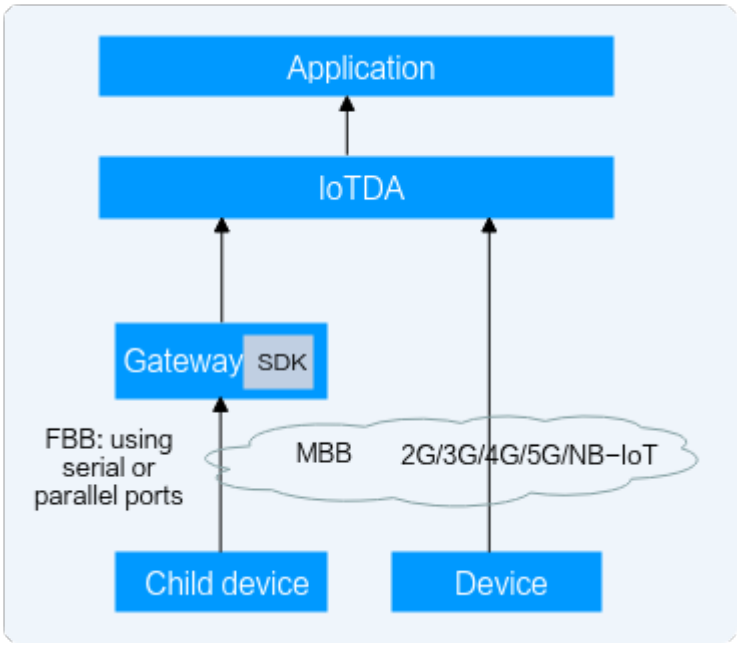


2.2.2.5 Child Devices

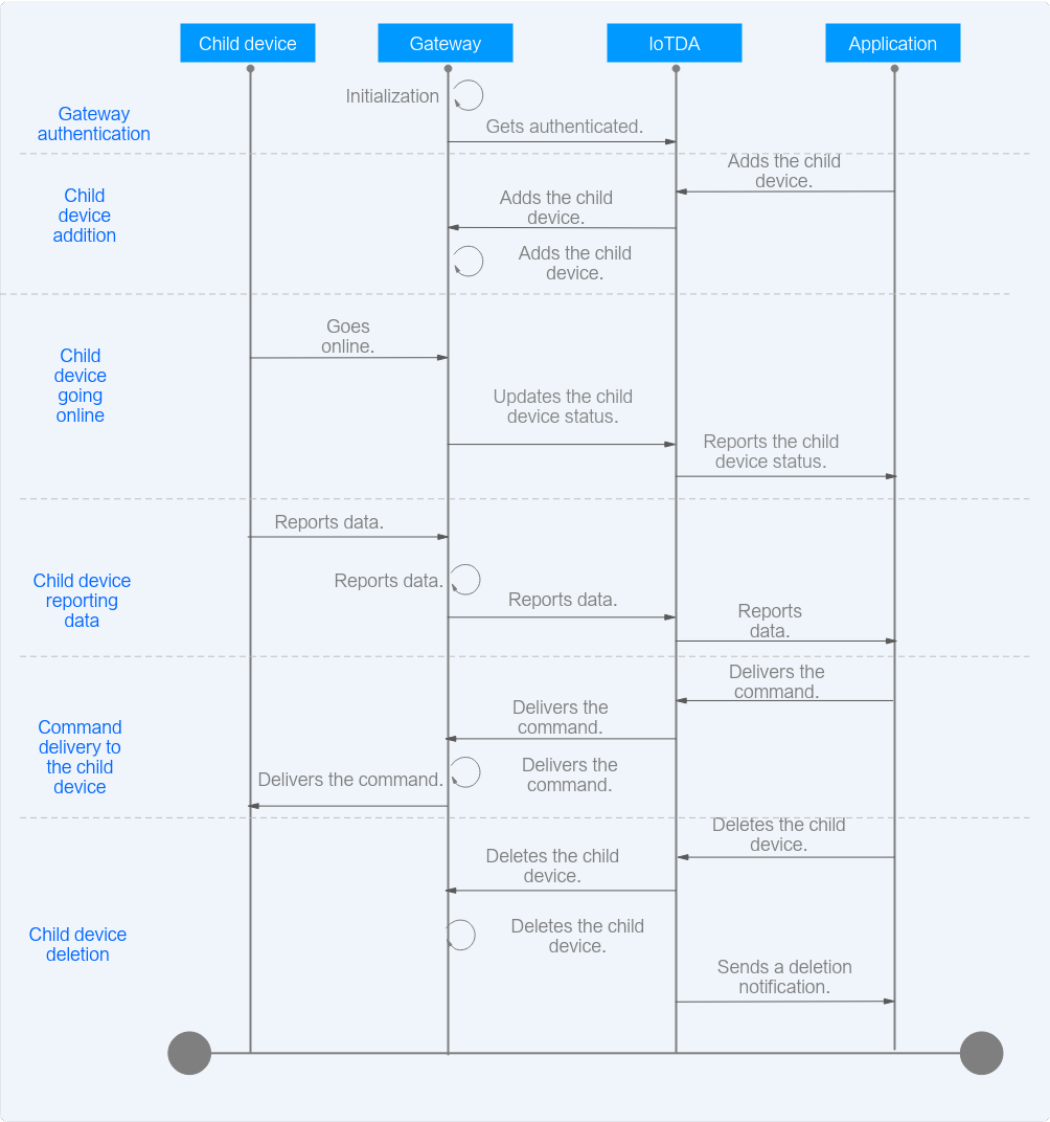
Overview

Devices can be directly or indirectly connected to IoTDA. Indirectly connected devices access IoTDA through gateways.

Devices that do not support TCP/IP cannot directly communicate with IoTDA, and need to use gateways as media.



Service Process



You can use the APIs provided by IoT Device SDKs to connect devices to IoTDA through gateways. API names of the SDK vary depending on the language. For details, see "Using IoT Device SDKs for Access" in *Developer Guide*.

1. Create a product and register a gateway on the platform.
2. The gateway calls the **Authentication** API to go online.
3. After the gateway is authenticated, you can **add child devices**. After the child device is added, you can **view it**.
4. The status of the newly added child device is still displayed as **Inactive** on the console. This is because the gateway has not reported the latest status of the child device to the platform. Call the API **Updating the Child Device Status** after the child device is added or before the child device reports data.

> **NOTE**
>
> The status of a child device indicates whether the child device is connected to the gateway, and the gateway reports the status to IoTDA for status updates. If the gateway does not report the status of a child device, the child device status is not updated on the platform. For example, after a child device connects to IoTDA through a gateway, the child device status is displayed as online. If the gateway is disconnected from IoTDA, the gateway can no longer report the child device status and IoTDA will consider the child device online.

5. The gateway calls the API **Batch Reporting Device Properties** to report the data of the child device. The parameters in the API request are the information about the gateway and the child device.
6. The gateway subscribes to a topic for command delivery, and receives and processes events delivered by the application or IoTDA platform.
7. The application calls the API **Deleting a Device** to deliver the event of deleting the child device to the gateway. The gateway deletes the device upon receiving the event.

> **NOTE**
>
> When an API is called to add or delete a child device on IoTDA, the platform delivers an operation event to the gateway. Then, the gateway processes the child device addition or deletion events. When a child device is added or deleted on IoTDA but the gateway is offline, the platform caches the event (for a maximum of one day) and delivers the event after the gateway goes online.

Connecting a Gateway to IoTDA

The device reports data to the gateway by integrating the SDK on the gateway, and then the gateway forwards the data to IoTDA. For details about how to connect the gateway to IoTDA, see "Development on the Device Side" > "Using IoT Device SDKs for Access" in *Developer Guide*.

Adding a Child Device

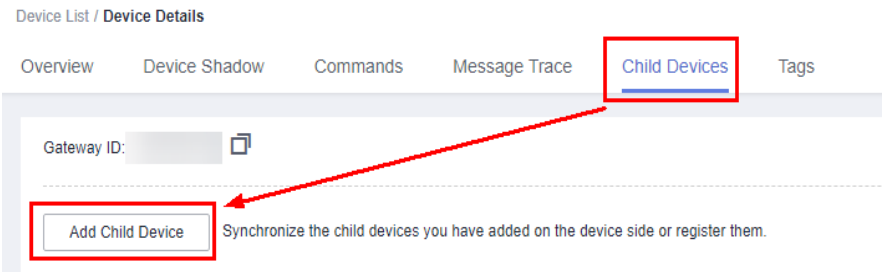
Method 1

After a gateway is connected to IoTDA, call the API **Creating a Device** to connect the child device to IoTDA.

Method 2

Step 1 Log in to the IoTDA console.

Step 2 In the left navigation pane, choose **Devices** > **All Devices**. On the device list, click **View** in the row of a gateway to access its details, and click the **Child Devices** tab > **Add Child Device**.



Step 3 Enter basic information and click **Yes**.

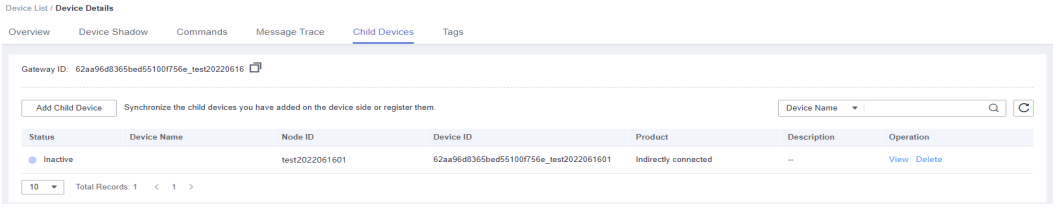
----End

Viewing a Child Device

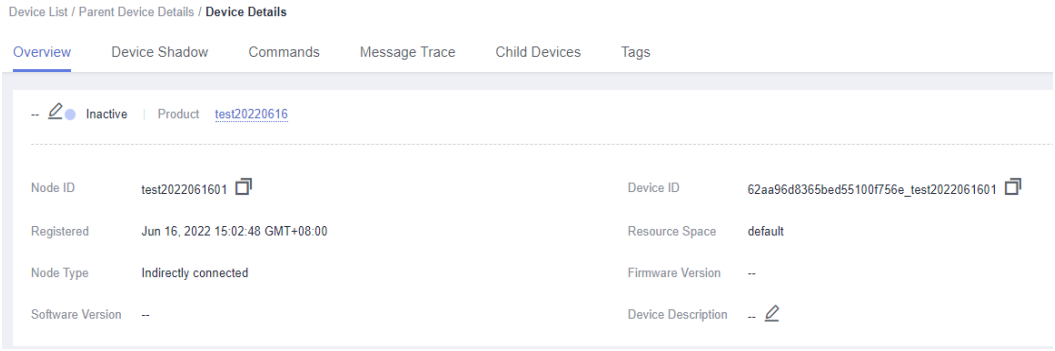
Step 1 Log in to the IoTDA console.

Step 2 In the left navigation pane, choose **Devices** > **All Devices**. On the device list, click **View** in the row of a gateway to access its details.

Step 3 On the **Child Devices** tab page, view the status, ID, and type of the child devices connected to IoTDA through the gateway.



Step 4 Click **View** in the row of a specific child device on the **Child Devices** tab page to view its **details**.



----End

2.2.2.6 Tags

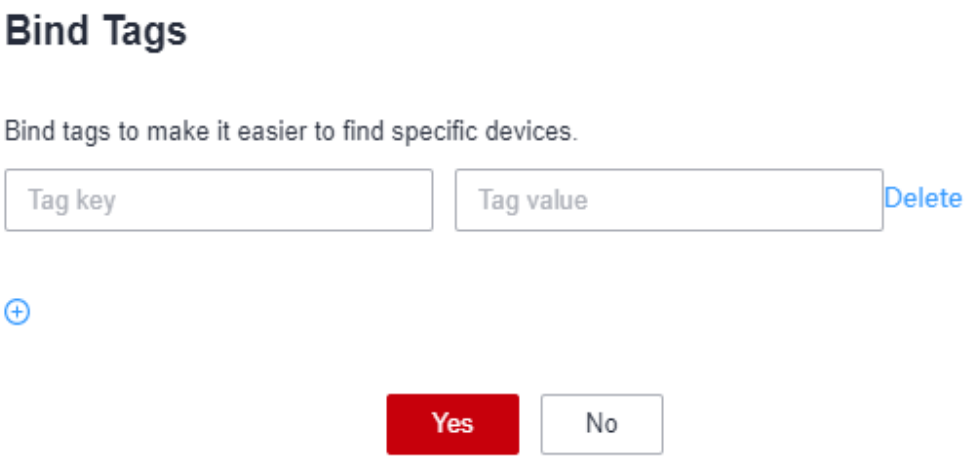
You can define tags and bind tags to devices. Tags are used to classify devices. You can bind tags to devices on the device details page.

Procedure

Step 1 Log in to the IoTDA console.

Step 2 In the left navigation pane, choose **Devices > All Devices**, and click **View** to open the device details page.

Step 3 On the **Tags** tab, click **Bind Tags** to bind one or more tags to the device.



----End

2.2.2.7 Asset Properties

Asset properties are used to manage assets. Asset properties of a device consist of a group of predefined fields. The field types can be **Number**, **String**, or **Boolean**. Restrictions on asset properties are defined by products. That is, products can predefine asset properties (including field names, types, and restrictions). When device asset properties are created or modified, they must match the asset properties defined by the product. Otherwise, the asset properties cannot be saved.

Procedure

Step 1 Log in to the IoTDA console.

Step 2 In the left navigation pane, choose **Products**. In the product list, click the name of the product to which your device belongs to go to the **Products** page.

Step 3 Click the **Asset Properties** tab, click **Add Field**, and set parameters based on Table 2-3.

Table 2-3 Adding a field

Parameter	Description
Name	Property name.
Alias	Property alias.

Parameter	Description
Type	Select a type. • **Number** – **Min Value**: minimum value of the number type. – **Max Value**: maximum value of the number type. • **String** – **Max Length**: maximum length of the string. – **Regular Expression**: used to verify the format of asset properties of the string type. – **Enumerated Values**: values available for **field_value** of asset properties of the string type. • **Boolean**: none.
Mandatory	By default, this parameter is disabled.
Available for Search	If you enable the search function, the asset property can be used for advanced search. **NOTE** • After adding a field, you need to edit the device asset property value again for the field to take effect in advanced search. • A maximum of five asset fields can be searched for at the same time for a single product.
Default Value	Customize a default value. **NOTE** After setting a default value, you need to edit the device asset property value again for the field to take effect in advanced search.

Step 4 In the left navigation pane, choose **Devices** > **All Devices**, click the device of your product to go to its details page.

Step 5 Click the **Asset Properties** tab to set the asset properties of the device.

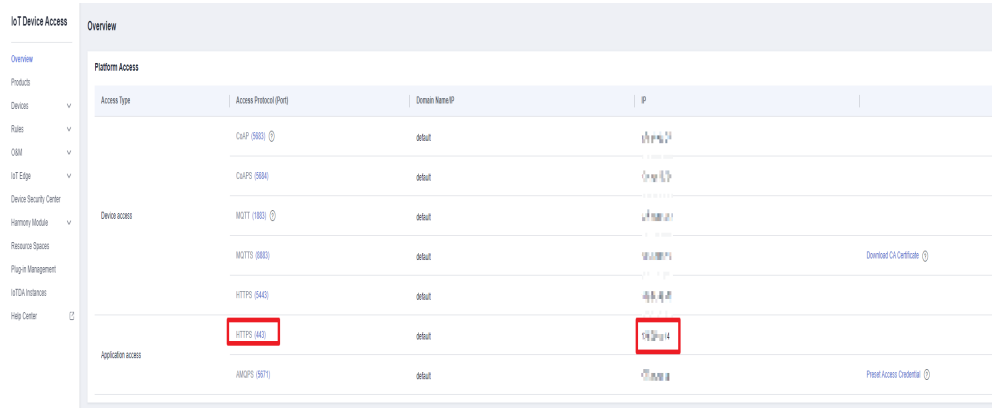
----End

2.2.2.8 Cloud O&M Configuration

IoTDA delivers O&M configuration to devices through device events, implementing collaborative management of device configuration on the cloud.

Step 1 Log in to the IoTDA console and obtain the application access address on the **Overview** page.

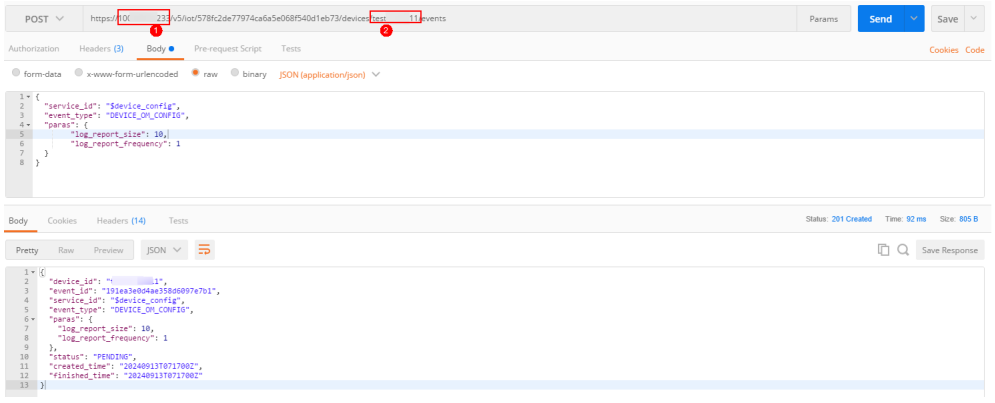
Figure 2-8 Querying the application access address



Step 2 Call the API to deliver O&M configuration through events (using Postman as an example). In the following example, *service_id* are set to **$device_config**, and *event_type* is set to **DEVICE_OM_CONFIG**. *ip* indicates the application access address, *device_id* indicates the ID of the target device, and *project_id* indicates the project ID.

```
uri: https://{ip}/v5/iot/{project_id}/devices/{device_id}/events
Request body:
{
  "service_id": "$device_config",
  "event_type": "DEVICE_OM_CONFIG",
  "paras": {
    "log_report_size": 10,
        "log_report_frequency": 1
  }
}
```

Figure 2-9 Delivering device O&M policies



----End

2.2.2.9 Device Configuration Detection

Devices report their configuration to IoTDA through property reporting, so that IoTDA can detect and manage device configuration.

Step 1 Report the device configuration value. Report the device configuration by referring to *IoT Device Access (IoTDA) 25.9.0.SPC002 Usage Guide (for Huawei Cloud Stack 8.6.0)* > *IoT Device Access (IoTDA) 25.9.0.SPC002 API Reference (for Huawei Cloud Stack 8.6.0)* > "API on the Device Side" > "Device Properties". In the following

example, **value1**, **value2**, and **value3** are sample values, and *$device_sys_config* is the service ID of the device configuration to be reported.

NOTE

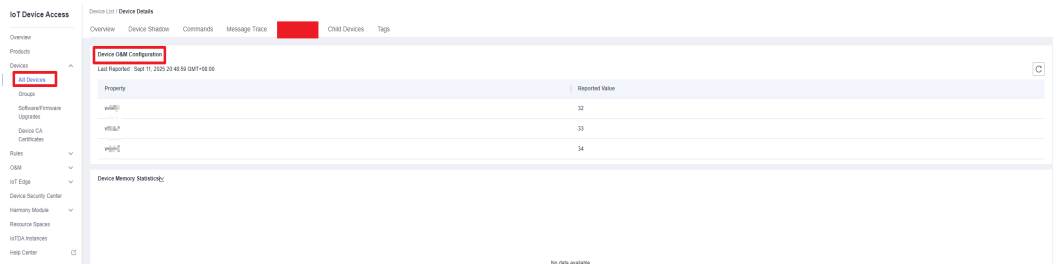
$device_sys_config is a system-level service ID. Distinguish it from the service ID defined in the product model.

Example of device configuration reporting:

```
{
   "services": [
      {
         "service_id": "$device_sys_config",
         "properties": {
            "value1": 32,
            "value2": 33,
            "value3": 34
         },
         "event_time": "20240826T103212Z"
      }
   ]
}
```

Step 2 Check the configuration reported by the device. In the navigation pane, choose **Devices** > **All Devices**. Click the **Device List** tab, click the device name to access its details page, and go to the maintenance page to check the configuration reported by the device.

Figure 2-10 Checking device configuration



----End

2.2.3 Groups

A group is a collection of devices. You can create groups for all the devices in a resource space based on different rules, such as regions and types, and you can operate the devices by group. For example, you can perform a firmware upgrade on a group of water meters in the resource space. Devices in a group can be added, deleted, modified, and queried. A device can be bound to and unbound from multiple groups.

Constraints

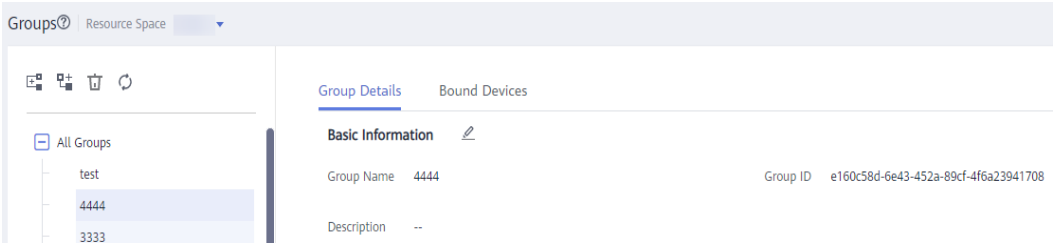
- A maximum of 20,000 devices can be added to a group.
- A device can be bound to a maximum of 10 groups.

Managing Groups

Step 1 Log in to the IoTDA console.

Step 2 In the navigation pane, choose **Devices** > **Groups**.

Step 3 Click the buttons to add or delete a group.



Icon	Description
	Add a root group. The group name and description can be customized.
	Add a child group. The group name and description can be customized.
	Delete a selected group. The deletion operation cannot be undone. Before deleting a group, unbind devices from the group and delete its lower-level groups.

----End

Binding or Unbinding a Device

After adding a group, you can bind a device to or unbind a device from the group.

Step 1 Log in to the IoTDA console.

Step 2 In the navigation pane, choose **Devices** > **Groups**.

Step 3 Select a group, click the **Bound Devices** tab, and click buttons to bind or unbind devices.

Icon	Description
Bind	Bind a device to a group.
Unbind	Unbind a device from a group.

----End

2.2.4 Software/Firmware Upgrades

2.2.4.1 Overview

Firmware is like a device driver for the hardware. It is responsible for the underlying work of a system, for example, the basic input/output system (BIOS) on a computer mainboard. Firmware upgrade, also called firmware over the air

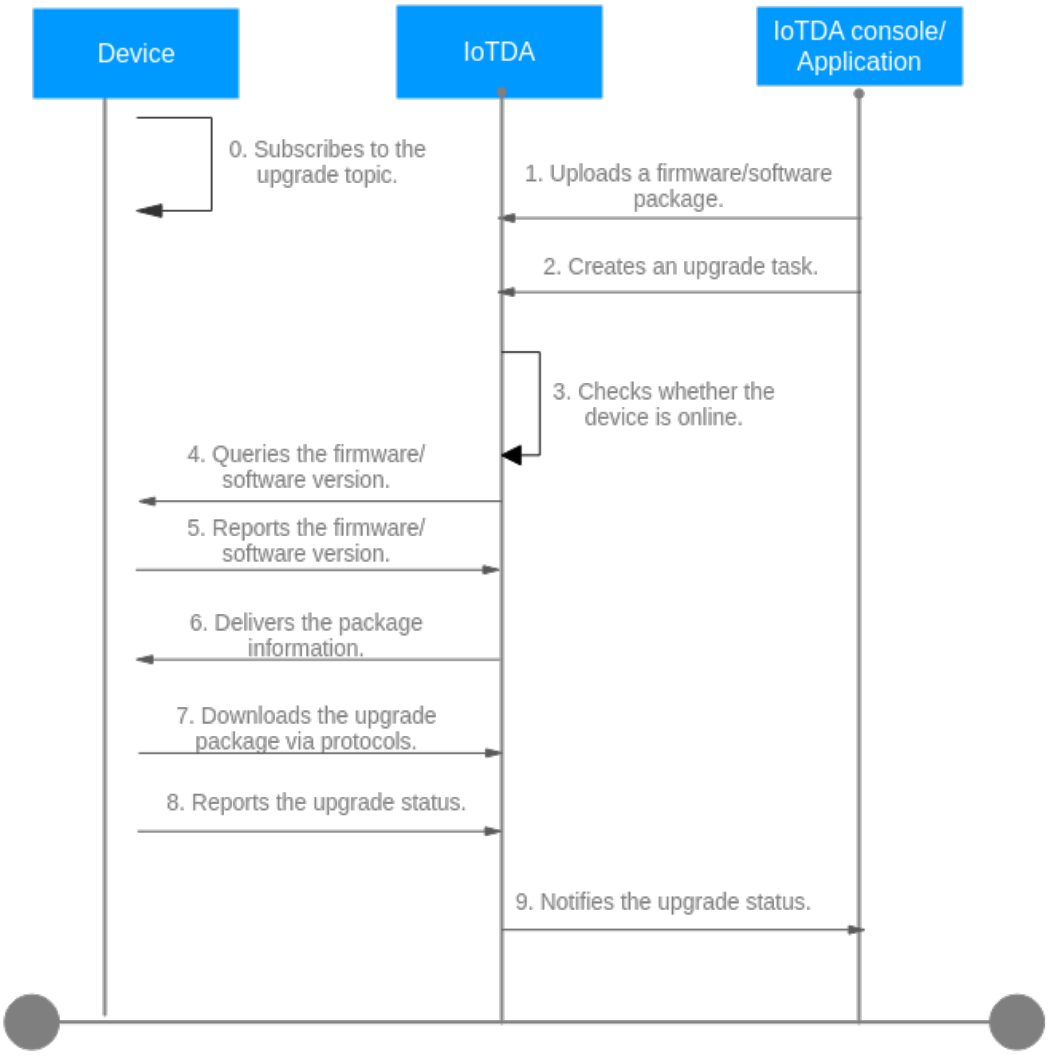
(FOTA), allows users to upgrade the firmware of MQTT or LwM2M devices in OTA mode.

Software includes system software and application software. The system software provides the basic device functions, such as the compilation tool and system file management. The application software provides functions such as data collection, analysis, and processing, depending on the features the device provides. Software upgrade, also called software over the air (SOTA), allows users to upgrade the software of MQTT devices in OTA mode. For an MQTT product model, the software upgrade protocol is not verified.

NOTE

Currently, software upgrades support only MQTT devices, and firmware upgrades support only MQTT or LwM2M devices.

Figure 2-11 Software/firmware upgrade for devices using MQTT



The FOTA upgrade process for devices using MQTT starts from the device subscribing to the upgrade topic. For details about the upgrade topic, see "API Reference on the Device Side" > "Software and Firmware Upgrade APIs" in *API Reference*. The main steps are as follows:

1. A user uploads a firmware or software package on the IoTDA console.
2. The user creates an upgrade task on the console or application server.
3. The platform checks whether the device is online and triggers the upgrade negotiation process immediately when the device is online. If the device is offline, the platform waits for the device to go online and subscribe to the upgrade topic. After detecting that the device goes online, the platform triggers the upgrade negotiation process.
4. The platform delivers a command to query the device version.
5. After the query is successful, the platform determines whether the device needs to be upgraded based on the target version.
   - If the returned version is the same as the target version, no upgrade is required. The upgrade task is marked successful.
   - If the returned version is different from the target version and this version supports upgrades, the platform continues the upgrade.
6. The platform delivers the package information.
7. The device downloads the upgrade package through the protocol.
   - If the download type is HTTPS, the platform delivers the URL, token, and package information. The user downloads the upgrade package using HTTPS based on the URL and token. The token is valid for 24 hours.
8. The device performs the upgrade. After the upgrade is complete, the device returns the upgrade result to the platform.
9. The platform notifies the console or application of the upgrade result.

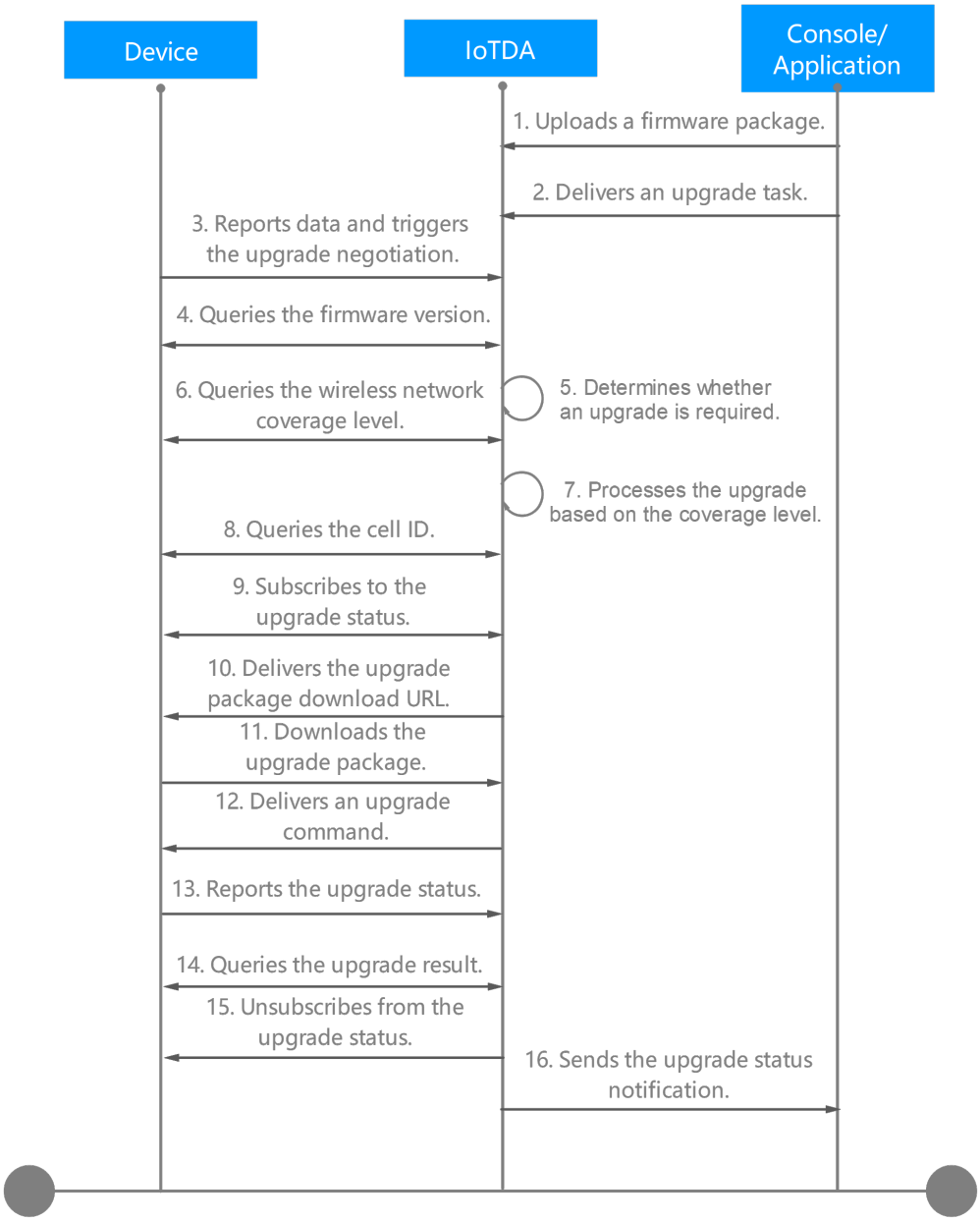
2.2.4.2 Firmware Upgrades

Overview

Firmware is like a device driver for the hardware. It is responsible for the underlying work of a system, for example, the basic input/output system (BIOS) on a computer mainboard.

Firmware upgrade, also called firmware over the air (FOTA), allows you to upgrade the firmware of LwM2M or MQTT devices in OTA mode.

Firmware Upgrade for Devices Using LwM2M



The firmware upgrade process for a device using LwM2M is as follows:

1. A user uploads a firmware package on the IoTDA console.
2. The user creates an upgrade task on the console or application server.
3. The LwM2M device reports data to the platform. The platform detects that the device is online and triggers the upgrade negotiation process. (The timeout interval is 24 hours.)
4. The platform delivers a command to the device to query the device firmware version.
5. After the query is successful, the platform determines whether the device needs to be upgraded based on the target version. (In **4**, the timeout interval for the device to report the firmware version is 3 minutes.)

- If the returned firmware version is the same as the target version, no upgrade is required.
- If the returned firmware version is different from the target version, the platform continues the upgrade.

6. The platform queries the radio coverage of the cell where the device resides, and obtains the cell ID, reference signal received power (RSRP), and signal to interference plus noise ratio (SINR). (The timeout interval for the reporting of the radio coverage level and cell ID is about 3 minutes.)
   - If the query is successful, the IoT platform calculates the number of concurrent upgrade tasks based on the following methods and continues with **9**.
     - RSRP and SINR in range 0: 50 devices in the cell can be upgraded simultaneously.
     - RSRP in range 0 and SINR in range 1: 10 devices in the cell can be upgraded simultaneously.
     - RSRP in range 1 and SINR in range 2: Only one device in the cell can be upgraded at a time.
     - RSRP and SINR can be queried but are not within any of the three ranges: Only one device in the cell can be upgraded at a time.

Radio Coverage Range	RSRP Range (dBm)	SINR Range (dB)
0	–105 ≤ RSRP	7 ≤ SINR
1	–115 ≤ RSRP < –105	–3 ≤ SINR < 7
2	–125 ≤ RSRP < –115	–8 ≤ SINR < –3

     NOTE

     If only a small number of devices can be upgraded simultaneously, you can contact the local carrier to see if coverage can be improved.
   - If the query fails, the process continues with **8**.
7. The operation proceeds according to the wireless coverage level.
8. The platform delivers a command to query the cell ID of the device.
   - If the query is successful, 10 devices in the cell can be upgraded simultaneously.
   - If the query fails, the upgrade fails.
9. The platform subscribes to the firmware upgrade status from the device.
10. The platform delivers the URL of the firmware package to the device.
11. The platform instructs the device to download the firmware package. The device downloads the firmware package from the URL. After the download is complete, the device notifies the platform. Firmware packages can be downloaded in segments, and resumable download is supported. (The timeout interval is 60 minutes.)
12. The platform delivers the upgrade command to the device. The device performs the upgrade.

13. After the upgrade is complete, the device notifies the platform. (The timeout interval for the device to report the upgrade result and status is 30 minutes.)
14. The platform delivers a command to query the firmware upgrade result.
15. After obtaining the upgrade result, the platform unsubscribes from the upgrade status notification from the device.
16. The platform notifies the console and application server of the upgrade result.

Uploading a Firmware Package

On the IoTDA console, you can upload a firmware package to the **Firmware List** page for management.

NOTE

You need to provide and directly upload the upgrade file delivered to the device. The platform does not restrict the content of the upgrade file.

Procedure:

Step 1 Log in to the IoTDA console.

Step 2 In the left navigation pane, choose **Devices** > **Software/Firmware Upgrades**.

Step 3 Choose **Manage Resource Package** > **Firmware List**, and click **Upload**.

Step 4 On the page displayed, set the parameters based on **Table 2-4**, and click **OK** to upload a firmware package.

Table 2-4 Uploading firmware

Parameter	Description
Firmware File	Add a firmware package. The firmware package name cannot contain the special characters: +/?%#|;&=. The size of the upgrade package cannot exceed 1 GB. ZIP, BIN, RAR, and BZ2 packages are supported by default.
Firmware Version	Version of the firmware package.
Product	Select the corresponding product model.
Download Protocol	Select the protocol for downloading the upgrade package.
Source Versions	Enter the version manually. To add multiple versions, press **Enter** after inputting one version, and then input the next. If you do not enter any version numbers, the device of any version can be upgraded. If you enter version numbers, the firmware version reported by the device must exactly match any of the entered versions. Otherwise, the upgrade fails.
Firmware Package Fragment Size	Size of each segment of the firmware package downloaded by the device, in bytes. The value ranges from 32 to 500. The default value is **500**.

Parameter	Description
Description	Description of the upgrade package.

----End

Upgrading the Firmware for a Batch of Devices

There are two ways to upgrade the firmware for a batch of devices:

1. The application calls the API **Creating a Batch Task** to create an upgrade task for a batch of devices.

   For details, see "API Reference on the Application Side > Batch Task APIs > Creating a Batch Task" in *API Reference*.

2. Create a firmware upgrade task on the IoTDA console.

   NOTE

   Batch firmware upgrade tasks are automatically deleted from the task list after 30 days.

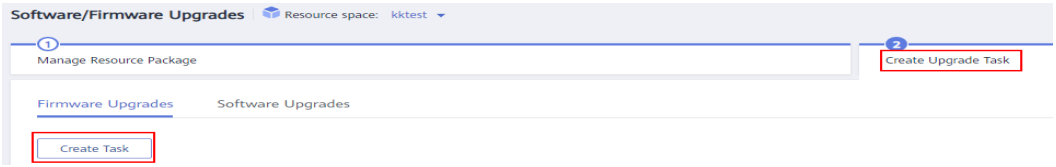
The following describes how to create a firmware upgrade task for a batch of devices on the console.

Procedure:

Step 1 Log in to the IoTDA console.

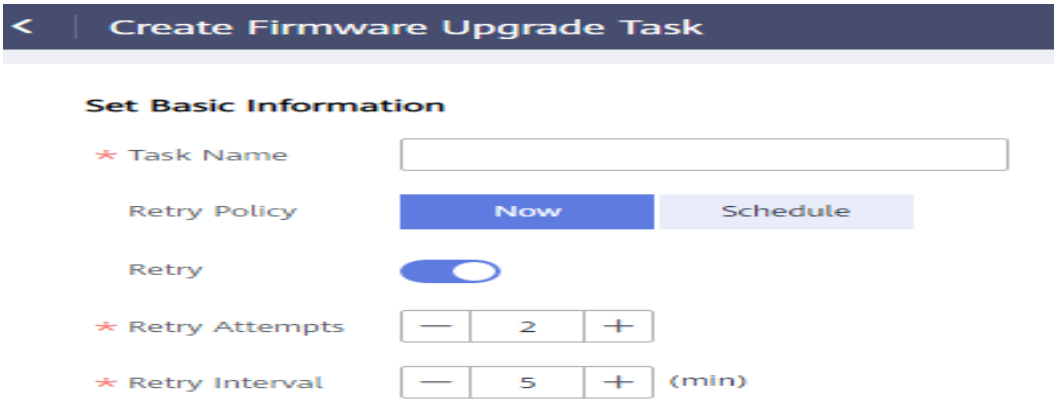
Step 2 In the left navigation pane, choose **Devices** > **Software/Firmware Upgrades**, and click **Create Upgrade Task**.

Step 3 On the **Firmware Upgrades** tab page, click **Create Task**.

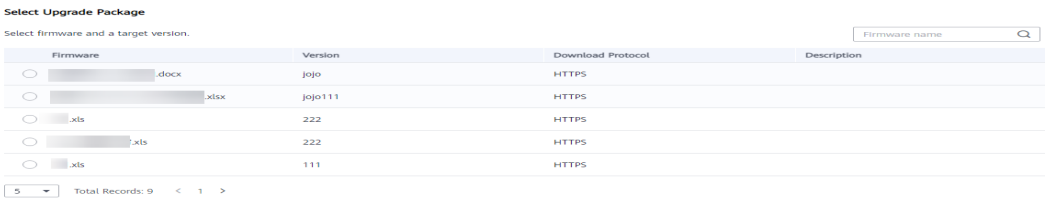


Step 4 Set the basic information, including the task name and retry policy, and enable the retry function.

If **Retry** is enabled, you can set the number of retry attempts and retry interval. You are advised to set **Retry Attempts** to 2 and **Retry Interval** to 5 minutes. If an upgrade fails, the upgrade will be retried 5 minutes later.

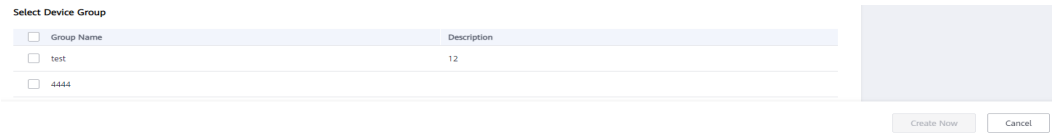


Step 5 Select a firmware package.

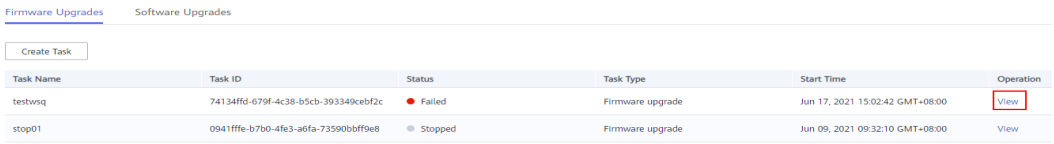


Step 6 Select the device group to upgrade and click **Create Now**.

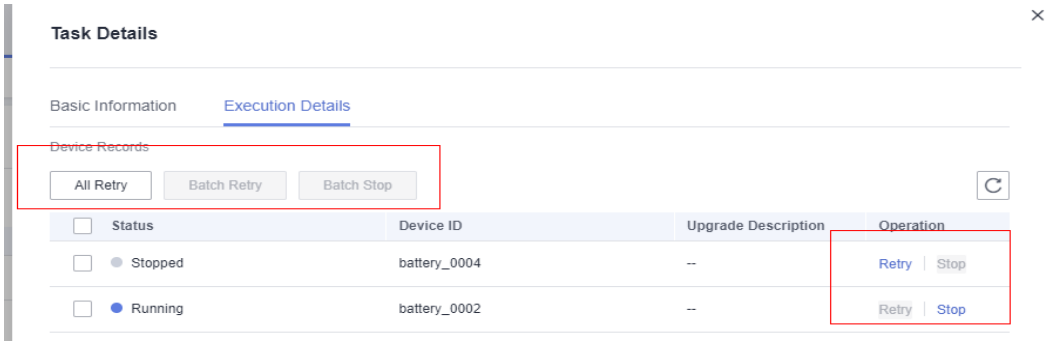
For details on how to create a group and add devices to the group, see **Groups**.



Step 7 View the result on the task list. You can click **View** to view the result for each device on the **Execution Details** page.



Step 8 On the **Execution Details** page, you can stop subtasks that are being processed or to be processed, and retry failed or stopped subtasks.



----End

2.2.4.3 Software Upgrade

Overview

Software includes system software and application software. The system software provides the basic device functions, such as the compilation tool and system file management. The application software provides functions such as data collection, analysis, and processing, depending on the features the device provides.

Uploading a Software Package

On the IoTDA console, you can upload a software package to the **Software List** page for management.

NOTE

You need to provide and directly upload the upgrade file delivered to the device. The platform does not restrict the content of the upgrade file.

Procedure:

Step 1 Log in to the IoTDA console.

Step 2 In the left navigation pane, choose **Devices** > **Software/Firmware Upgrades**.

Step 3 Choose **Manage Resource Package** > **Software List**, and click **Upload**.

Step 4 On the page displayed, set the parameters based on **Table 2-5**, and click **OK** to upload a software package.

Table 2-5 Uploading a software package

Parameter	Description
Software File	Add a software package. The software package name cannot contain the special characters: +/?%#|;&=. The size of the upgrade package cannot exceed 1 GB. ZIP, BIN, RAR, and BZ2 packages are supported by default.
Software Version	Version of the software package.
Product	Select the corresponding product model.
Download Protocol	Select the protocol for downloading the upgrade package.
Source Versions	Enter the version manually. To add multiple versions, press **Enter** after inputting one version, and then input the next. If you do not enter any version numbers, the device of any version can be upgraded. If you enter version numbers, the software version reported by the device must match any of the entered versions. Otherwise, the upgrade fails. Wildcards (* and ?) are supported.
Software Package Segment Size	Size of each segment of the software package downloaded by the device, in bytes. The value ranges from 32 to 500. The default value is **500**.
Description	Description of the upgrade package.

----End

Upgrading the Software for a Batch of Devices

There are two ways to upgrade the software for a batch of devices:

1. The application calls the API **Creating a Batch Task** to create an upgrade task for a batch of devices.

   For details, see "API Reference on the Application Side > Batch Task APIs > Creating a Batch Task" in *API Reference*.

2. Create a software upgrade task on the IoTDA console.

   NOTE

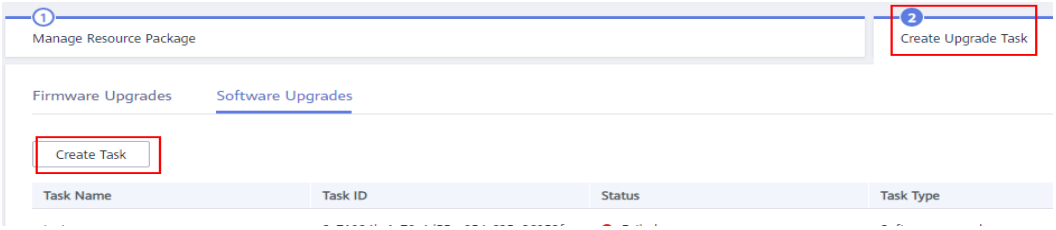
   Batch software upgrade tasks are automatically deleted from the task list after 30 days.

The following describes how to create a software upgrade task for a batch of devices on the console.

Step 1 Log in to the IoTDA console.

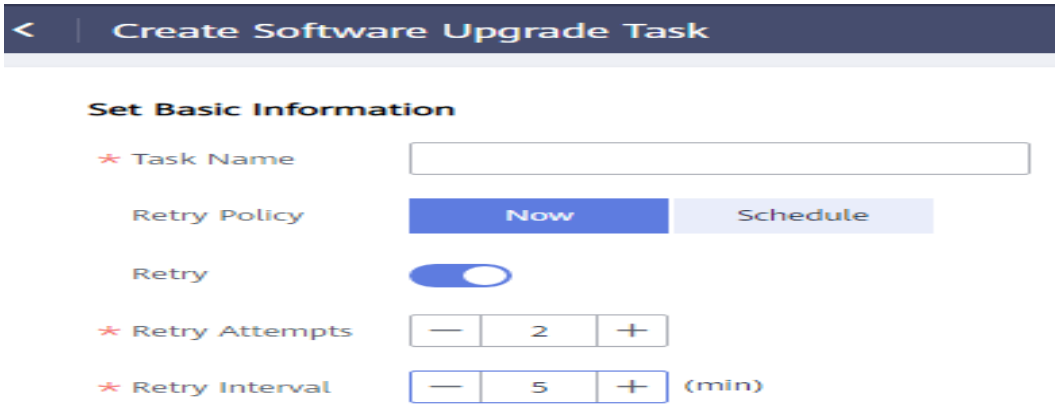
Step 2 In the left navigation pane, choose **Devices** > **Software/Firmware Upgrades**, and click **Create Upgrade Task**.

Step 3 On the **Software Upgrades** tab page, click **Create Task**.

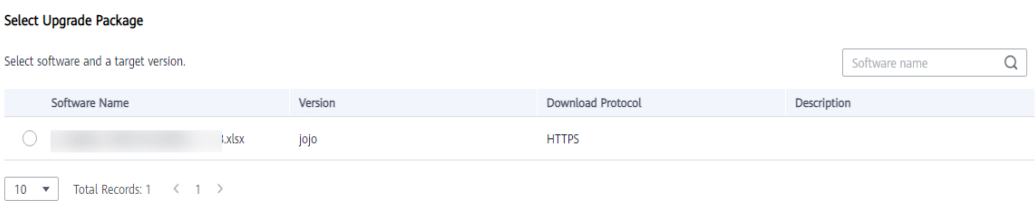


Step 4 Set the basic information, including the task name and retry policy, and enable the retry function.

If **Retry** is enabled, you can set the number of retry attempts and retry interval. You are advised to set **Retry Attempts** to 2 and **Retry Interval** to 5 minutes. If an upgrade fails, the upgrade will be retried 5 minutes later.

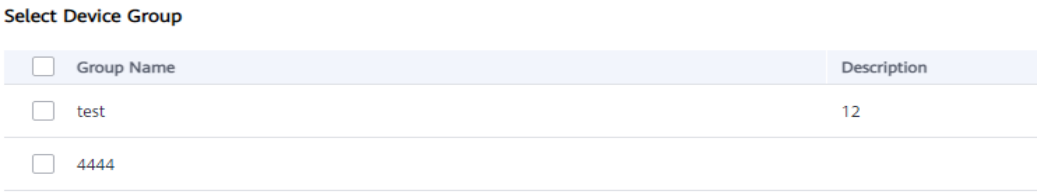


Step 5 Select a software package.

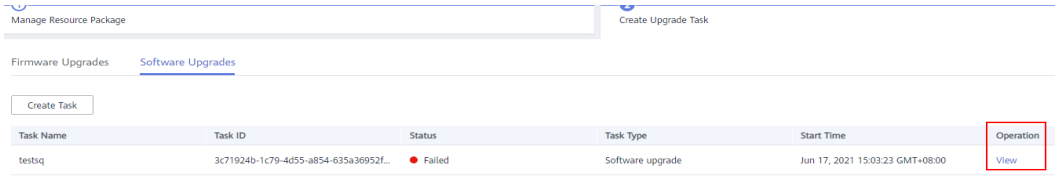


Step 6 Select the device or device group to upgrade and click **Create Now**.

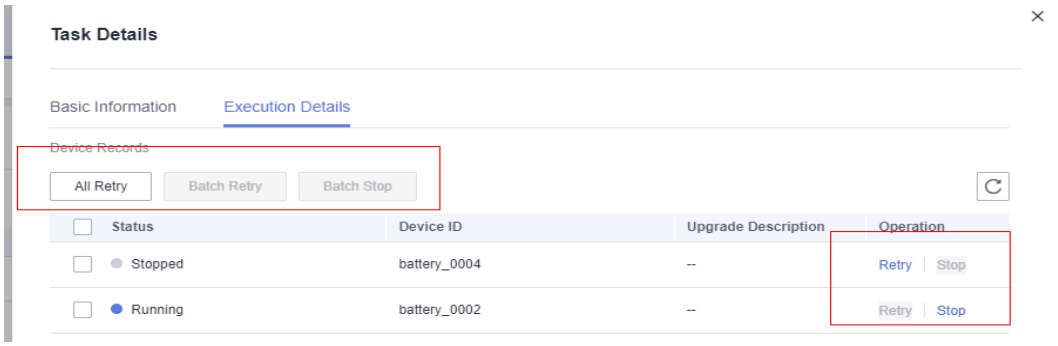
For details on how to create a group and add devices to the group, see **Groups**.



Step 7 View the result on the task list. You can click **View** to view the result for each device on the **Execution Details** page.



Step 8 On the **Execution Details** page, you can stop subtasks that are being processed or to be processed, and retry failed or stopped subtasks.

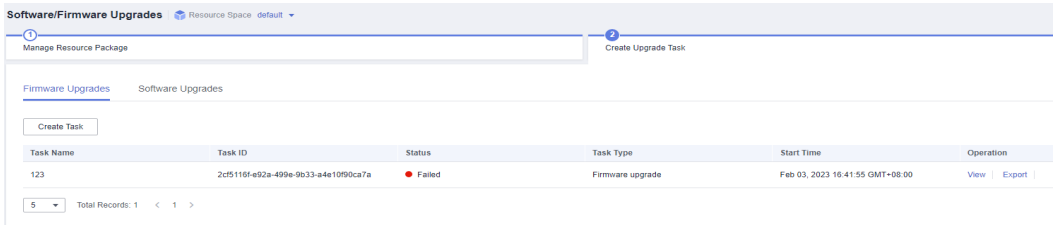


----End

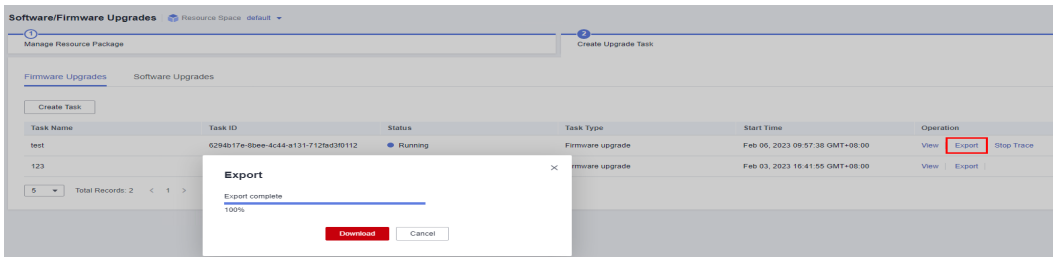
2.2.4.4 Exporting Software/Firmware Upgrade Details

You can export details about a software/firmware upgrade task to an XLS file on IoTDA. For details about restrictions on export tasks, see **Exporting Device Information**. The following describes how to export details about a firmware upgrade task.

Step 1 In the left navigation, choose **Devices** > **Software/Firmware Upgrades**, and click **Create Upgrade Task**. The upgrade task list page is displayed.



Step 2 Click **Export** to export the upgrade task details to an Excel file. After the export task is complete, download the file.



----End

2.2.5 Device CA Certificates

An X.509 certificate is a digital certificate used for communication entity authentication. IoTDA allows devices to use their X.509 certificates for authentication.

The use of X.509 certificate authentication protects devices from being spoofed. Before registering a device authenticated by an X.509 certificate, upload the device CA certificate to IoTDA and bind the device certificate to the device during device registration. This topic describes how to upload a device CA certificate to IoTDA.

Limitations

- Only MQTT devices can use X.509 certificates for identity authentication.
- A maximum of 100 device CA certificates can be uploaded.
- Only Base64-encoded .pem and .crt certificates are supported.

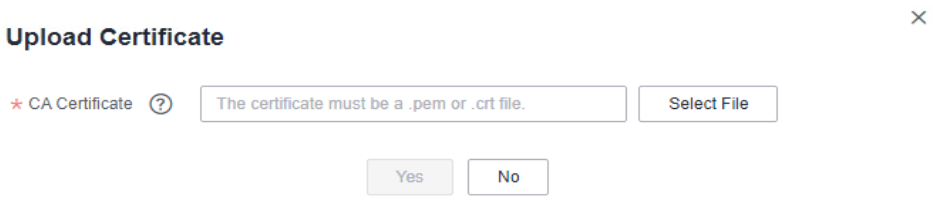
Uploading a Device CA certificate

Step 1 Log in to the IoTDA console.

Step 2 In the left navigation pane, choose **Devices** > **Device CA Certificates**, and click **Upload Certificate** in the upper right corner.

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

Figure 2-12 Uploading a certificate



NOTE

Device CA certificates are provided by device vendors. You can **create a commissioning certificate** during commissioning. For security reasons, you are advised to replace the commissioning certificate with a commercial certificate during commercial use.

----End

Making a Device CA Commissioning Certificate

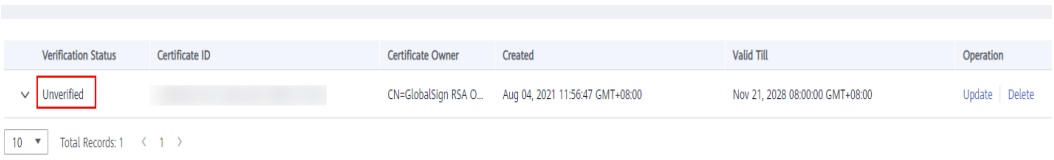
This section uses the Windows operating system as an example to describe how to use OpenSSL to make a commissioning certificate. The generated certificate is in PEM format and the suffix is **.cer**.

1. Download and install **OpenSSL**.
2. Open the CLI as user **admin**.
3. Run **cd c:\openssl\bin** (replace **c:\openssl\bin** with the actual OpenSSL installation directory) to access the OpenSSL view.
4. Generate a public/private key pair.
   ```
   openssl genrsa -out rootCA.key 2048
   ```
5. Use the private key in the key pair to generate a CA certificate.
   ```
   openssl req -x509 -new -nodes -key rootCA.key -sha256 -days 1024 -out rootCA.pem
   ```
   The system prompts you to enter the following information. All the parameters can be customized.
   - Country Name (2 letter code) [AU]: country, for example, CN
   - State or Province Name (full name) []: state or province, for example, GD
   - Locality Name (for example, city) []: city, for example, SZ
   - Organization Name (for example, company) []: organization
   - Organizational Unit Name (for example, section) []: organization unit, for example, IoT
   - Common Name (e.g. server FQDN or YOUR name) []: common name, for example, zhangsan
   - Email Address []: email address

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

Uploading a Verification Certificate

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



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

Step 1 Generate a key pair for the verification certificate.

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

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

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

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

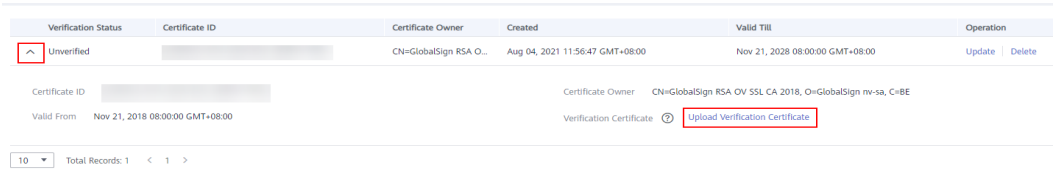
- Country Name (2 letter code) [AU]: country, for example, CN
- State or Province Name (full name) []: state or province, for example, GD
- Locality Name (for example, city) []: city, for example, SZ
- Organization Name (for example, company) []: organization.
- Organizational Unit Name (for example,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

Step 3 Use the CSR to create a verification certificate.

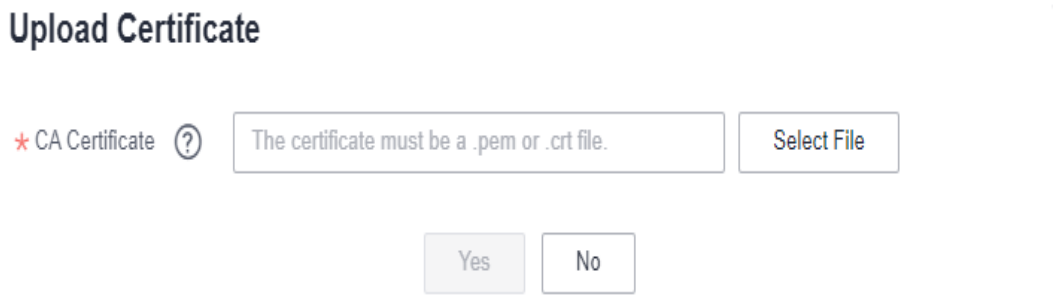
```
openssl x509 -req -in verificationCert.csr -CA rootCA.pem -CAkey rootCA.key -CAcreateserial -out verificationCert.pem -days 500 -sha256
```

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

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



Step 5 In the displayed dialog box, click **Select File** to add a file, and then click **Yes**.



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

----End

Presetting an X.509 Certificate

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

NOTE

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

Creating an X.509 Commissioning Certificate

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

2.2.6 MQTT X.509 Certificate Access

IoTDA supports MQTT X.509 certificate access. The following describes how to use an X.509 certificate to access IoTDA.

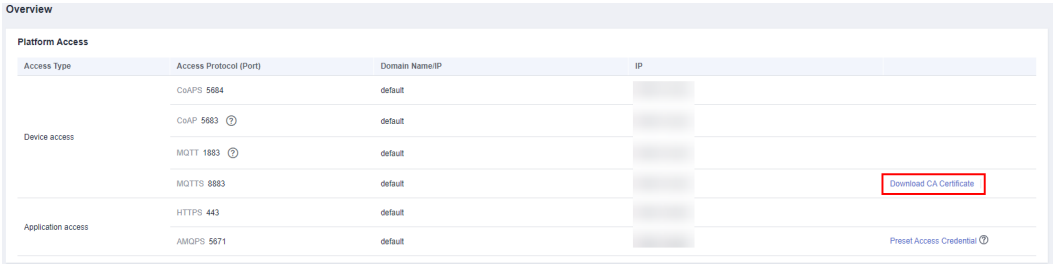
Preparations

Step 1 Log in to the IoTDA console.

Step 2 Upload a **device CA certificate** and **verification certificate**.

Step 3 Obtain the device certificate for commissioning by referring to **Presetting an X.509 Certificate**.

Step 4 Choose **Overview** in the left navigation pane, view the MQTTS access address and port, and click **Download CA Certificate** to save the server certificate file.



----End

Certificate materials obtained in the preceding steps are as follows:

- deviceCert.key: private key of the device certificate
- deviceCert.pem: device certificate
- mqtts-cert.pem: server CA certificate

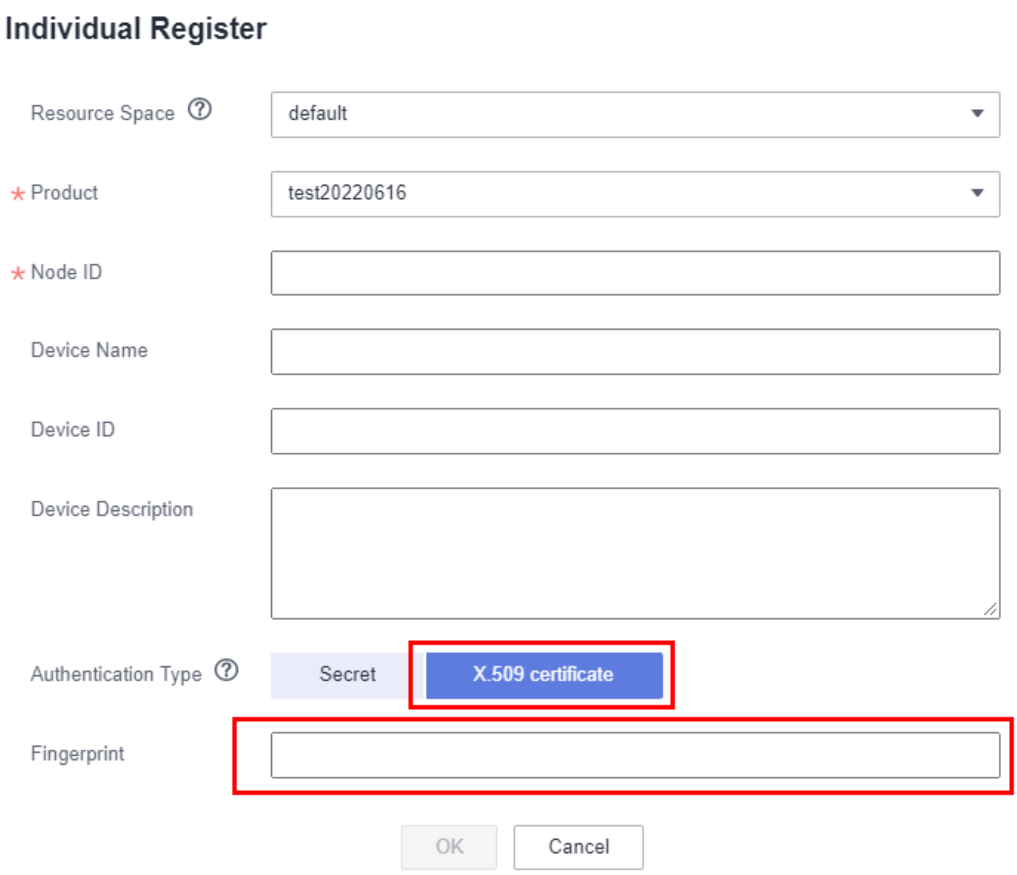
Registering an X.509 Certificate for Device Access

Step 1 When **creating a product**, select **MQTT** as the protocol.

Step 2 In the left navigation pane, choose **Devices** > **All Devices**, and click **Register Device** on the right.

Step 3 Enter the device registration information, and set **Authentication Type** to **X.509 certificate**. Run the following command to obtain the certificate fingerprint on the Linux platform:

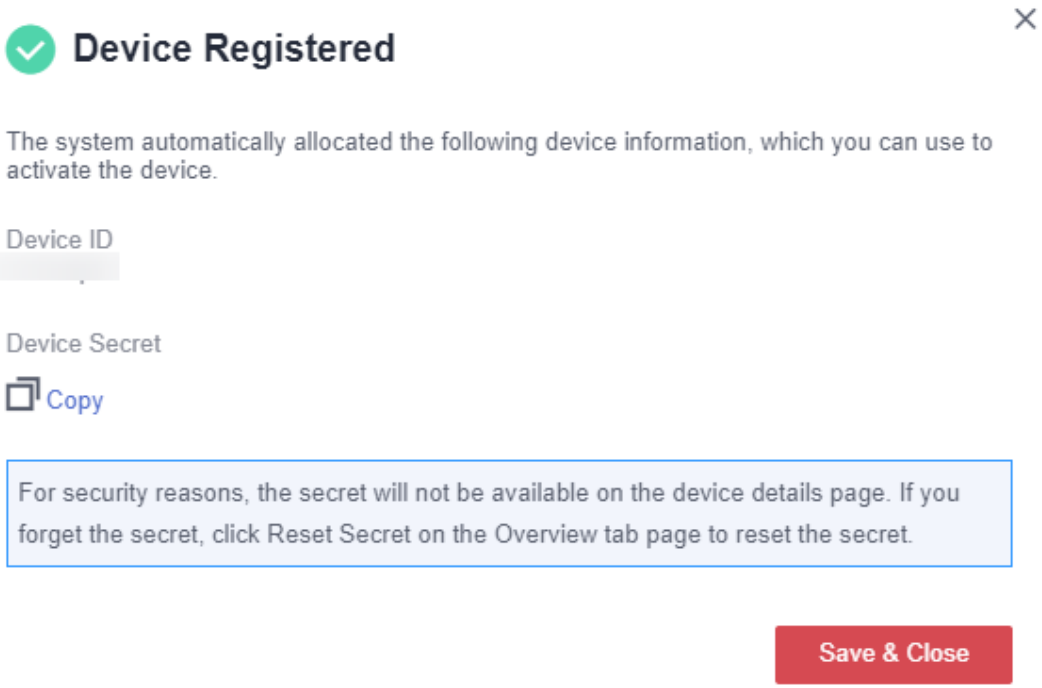
```
openssl x509 -fingerprint -sha256 -in client.crt | head -1 | awk '{print substr($2,13)}' | awk 'BEGIN {FS=":"; OFS=""} {for(i=1;i<=NF;++i) {out = out OFS $i}} END {print out;}'
```

NOTE

Fingerprint is optional for registering an X.509 certificate. The system uses the certificate of the device that is connected to the system for the first time. The device can use only the certificate of the device that is connected to the system for the first time.

Step 4 Click **Yes** to complete the device registration. Save the device ID for future use.

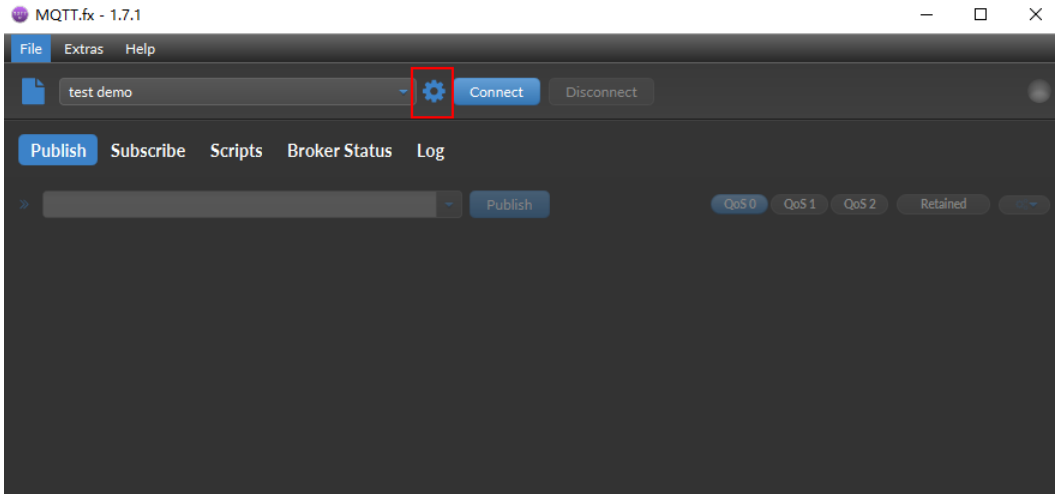


----End

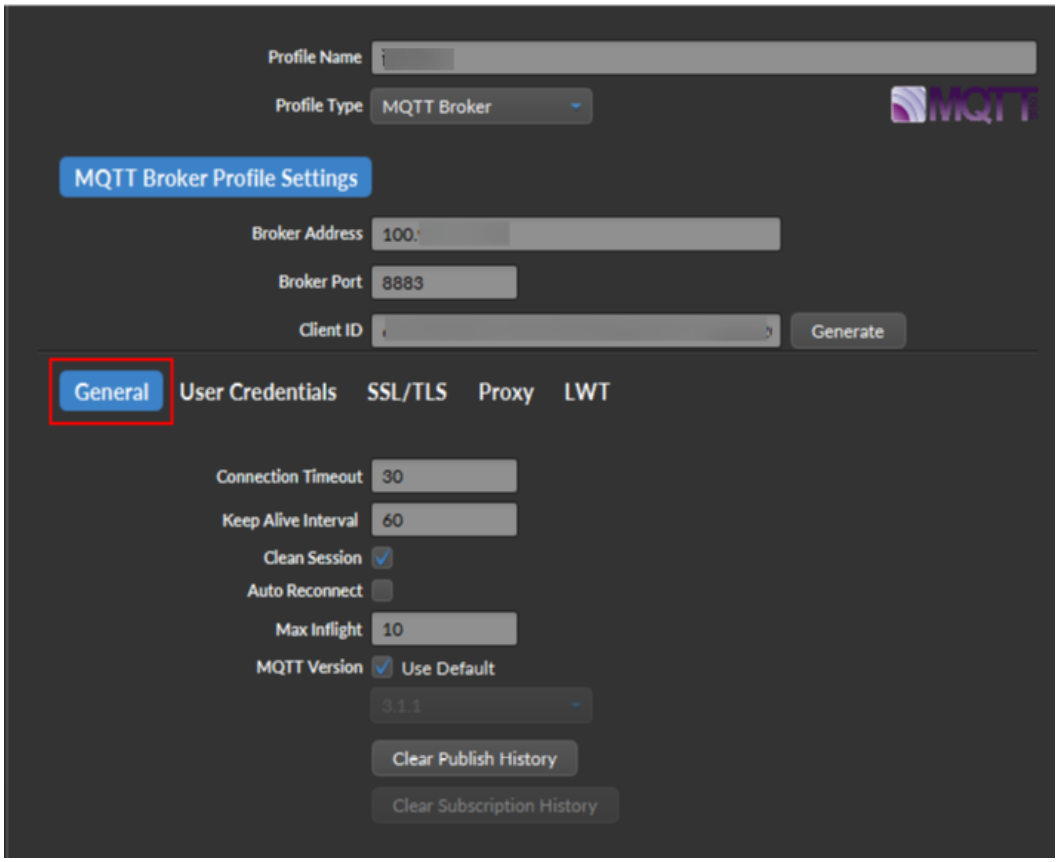
Device Access with Certificate

The following uses MQTT.fx as an example to describe how to connect a device using the certificate.

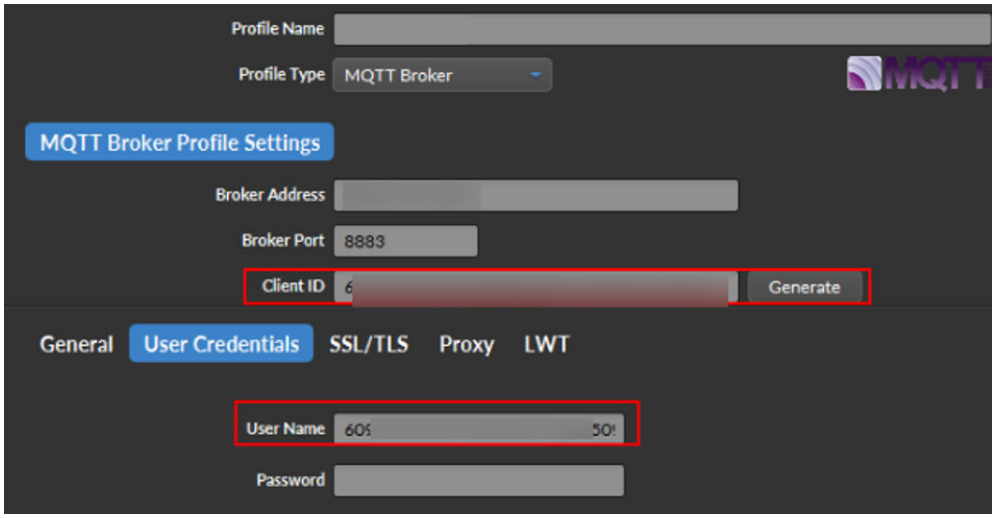
Step 1 Start the MQTT.fx client and click the setting icon.



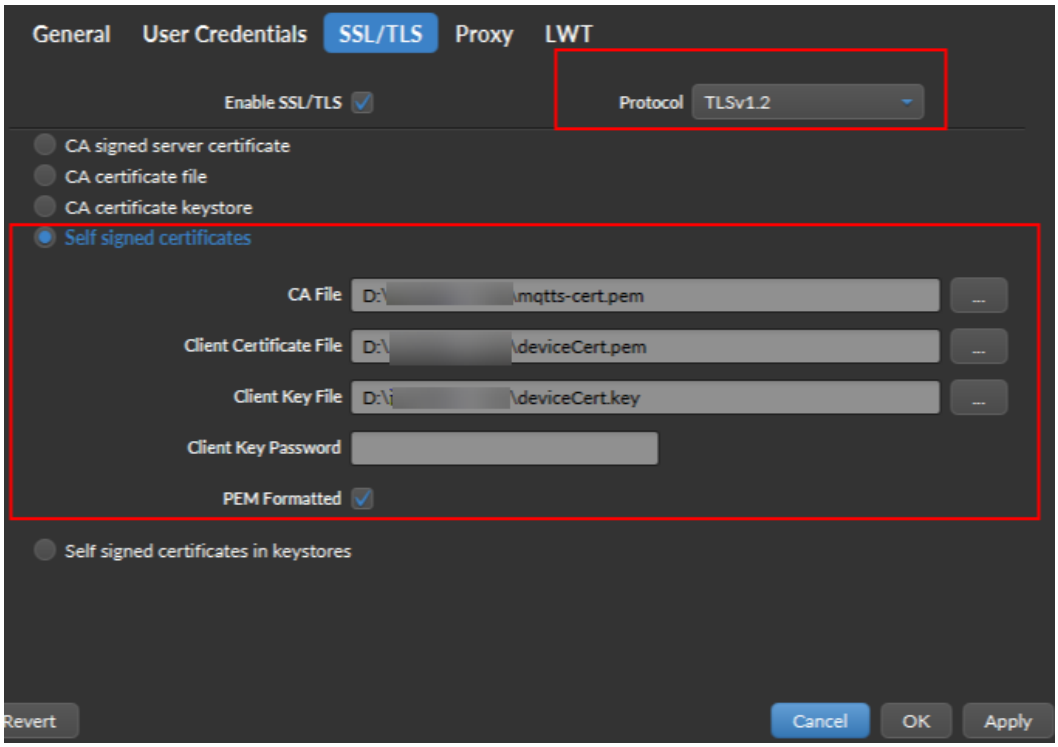
Step 2 Enter the information about the connection profile and the general information. You can retain the default values for **General**.



Step 3 Enter the user credential information. Set **Client ID** and **User Name** to the device ID obtained during device registration. You do not need to set **Password**.



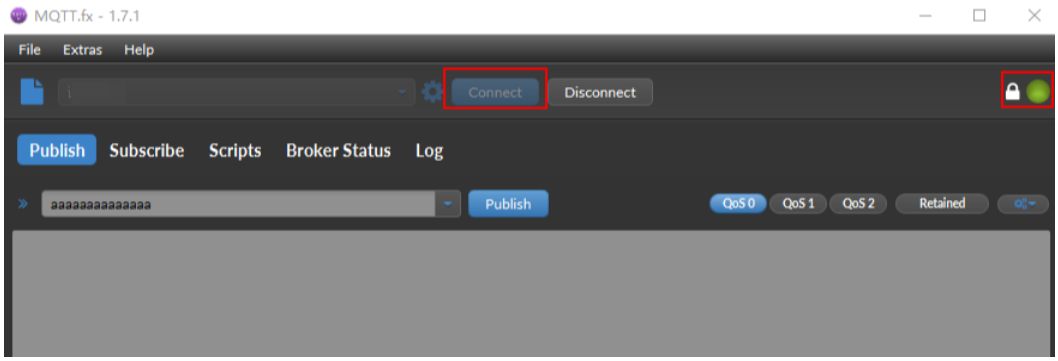
Step 4 Enable SSL/TLS, select the **Self signed certificates** check box, set **Protocol** to **TLSv1.2**. Configure the certificate materials obtained in **Preparations** as follows and click **OK**.



NOTE

- **CA File** indicates the device CA certificate.
- **Client Certificate File** indicates the device certificate.
- **Client Key File** indicates the private key of the device certificate.

Step 5 Return to the homepage, select the profile you just configured, and click **Connect**. If the status icon on the right is green, the device is successfully connected to the platform.



----End

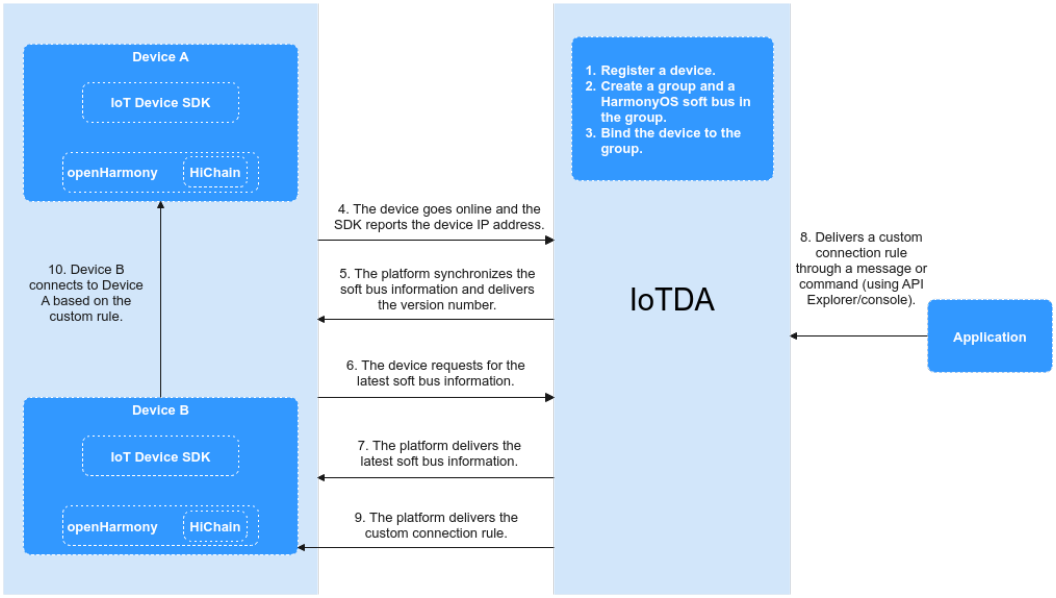
2.2.7 HarmonyOS Device Management

2.2.7.1 Device Connection Based on HarmonyOS Soft Bus

Overview

With the popularization of HarmonyOS devices, more and more users expect secure communications between devices based on the HarmonyOS technology. This section describes the management of HarmonyOS security groups and authorization identifiers for communications between HarmonyOS devices. The workflow is as follows:

Figure 2-13 Overall process



Constraints

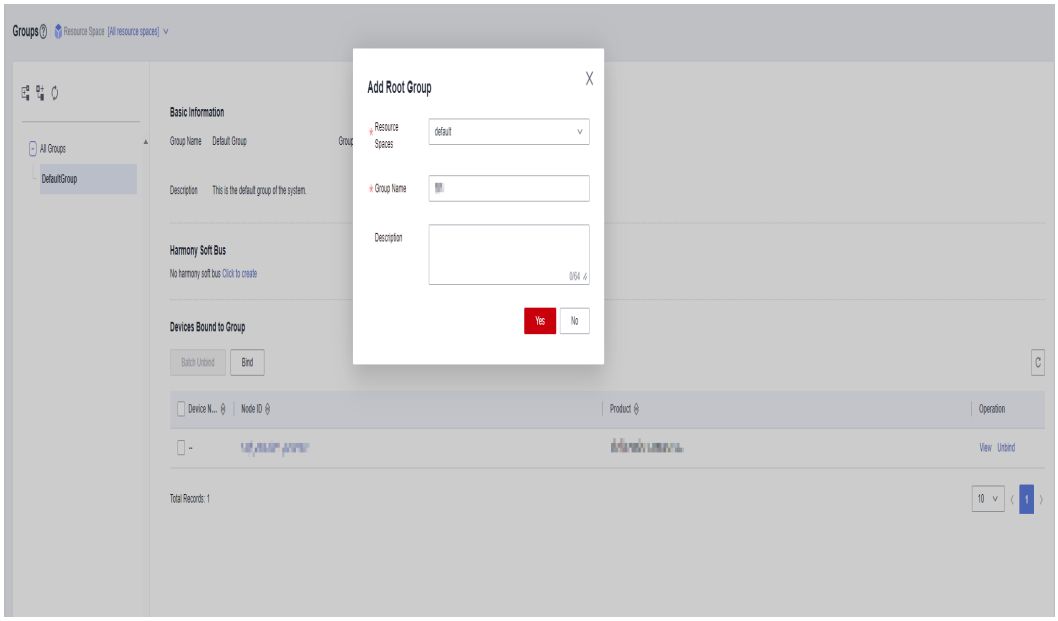
1. Each HarmonyOS soft bus group can contain up to 100 devices.
2. A device can be bound to up to 10 soft bus groups.

Procedure

Step 1 Log in to the IoTDA console.

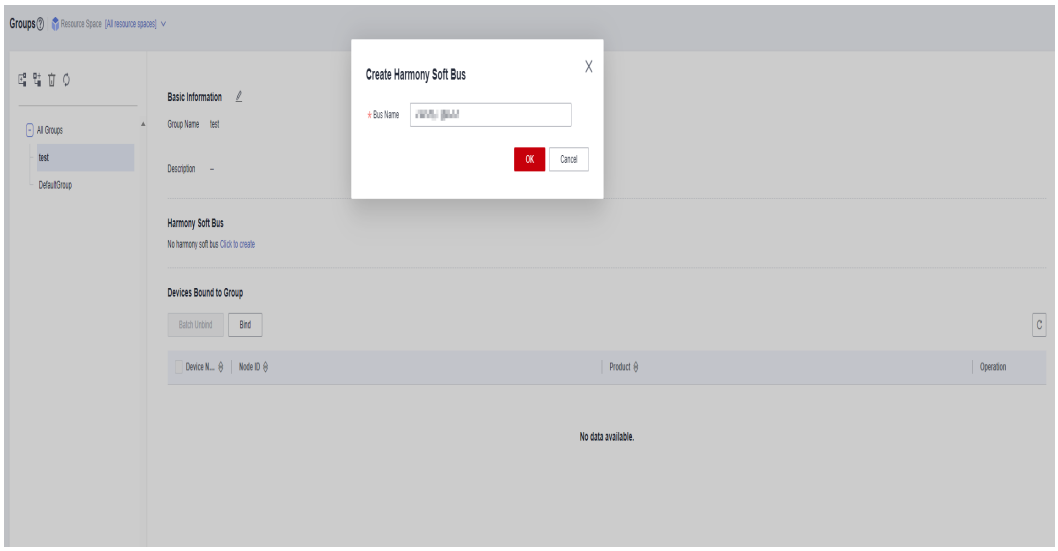
Step 2 In the navigation pane on the left, choose **Devices** > **Groups** and click **Add Root Group**.

Figure 2-14 Adding a root group



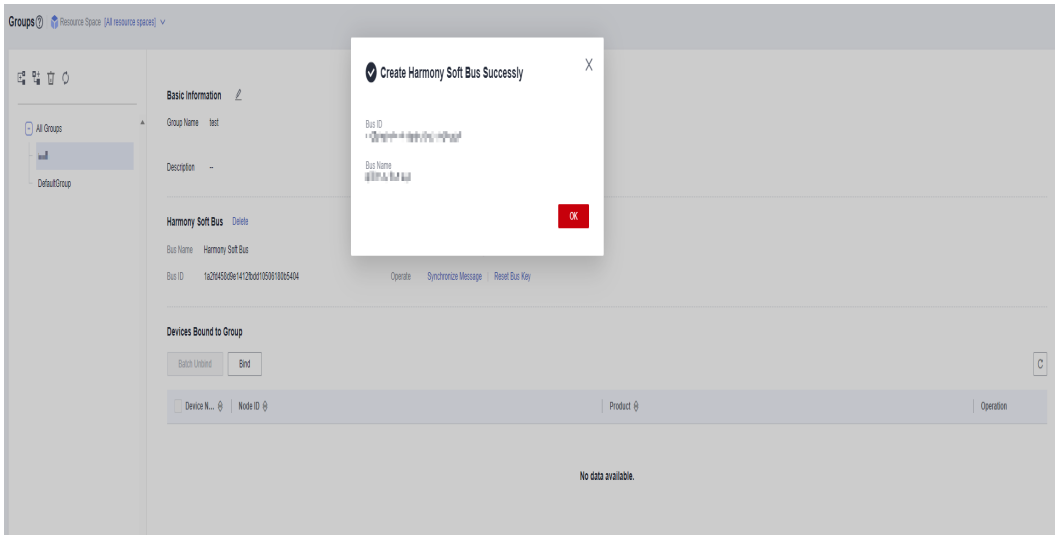
Step 3 Select the created group and create a HarmonyOS soft bus group.

Figure 2-15 Creating a HarmonyOS soft bus



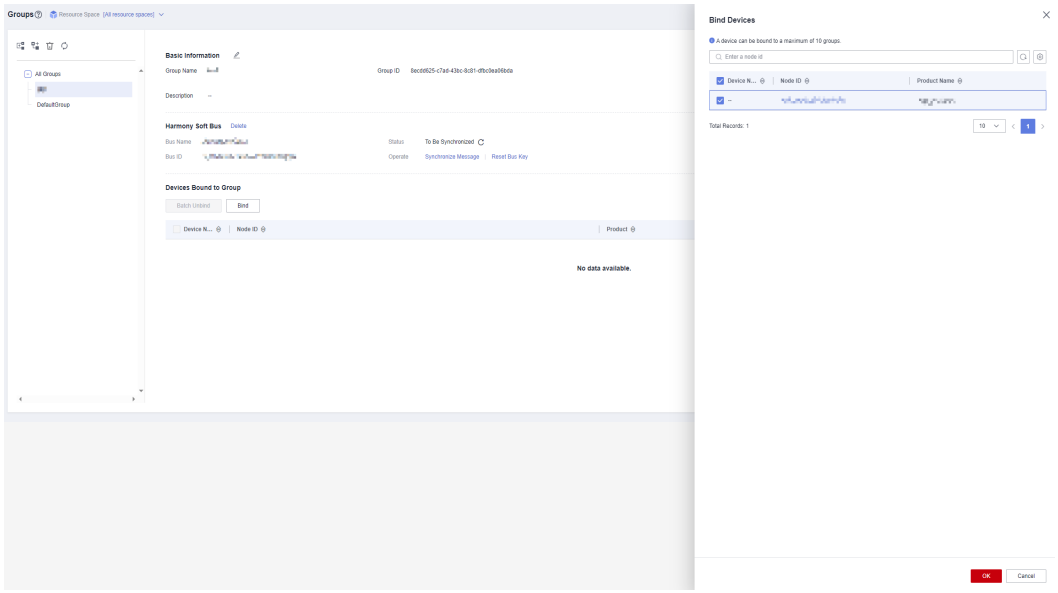
Step 4 After the soft bus is created, view the information indicating that the soft bus is created.

Figure 2-16 HarmonyOS soft bus information



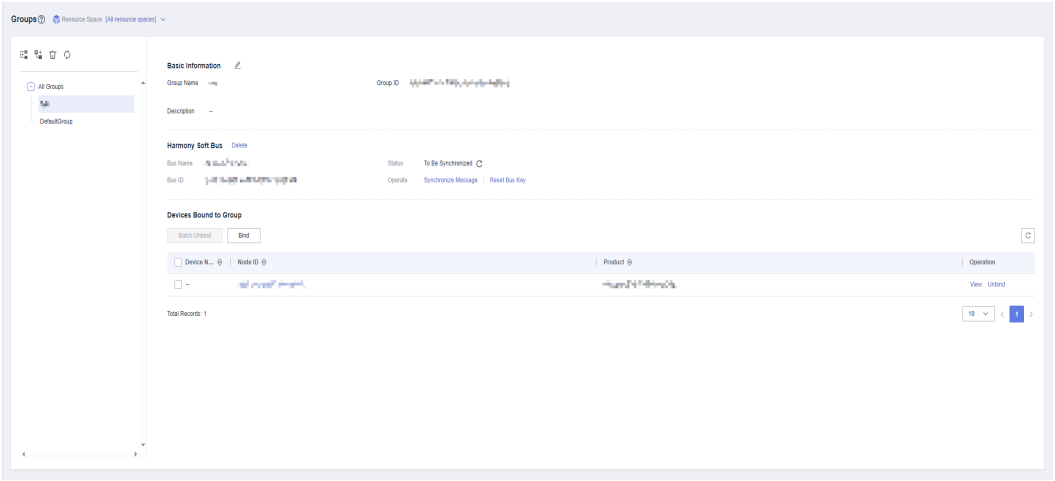
Step 5 Click **Bind**, select the device to be bound, and click **OK**.

Figure 2-17 Binding a device



Step 6 After the binding is complete, view the bound device in the list.

Figure 2-18 Checking bound devices



Step 7 After the device is bound, bring the device online for it to report the device IP address using the SDK.

NOTE

The device SDK reports the device IP address using the API described in chapter "Device Reporting Information" in the *API Reference*. The **device_ip** parameter is added to the **paras** parameter of this API and carries the IP address. For example, the **paras** parameter is as follows:

"paras": {

"device_sdk_version": "C_v0.5.0",

"sw_version": "v1.0",

"fw_version": "v1.0",

"device_ip": "127.0.0.1"

}

Step 8 On the console, click **Synchronize Message** to synchronize the HarmonyOS soft bus version information to the device.

NOTE

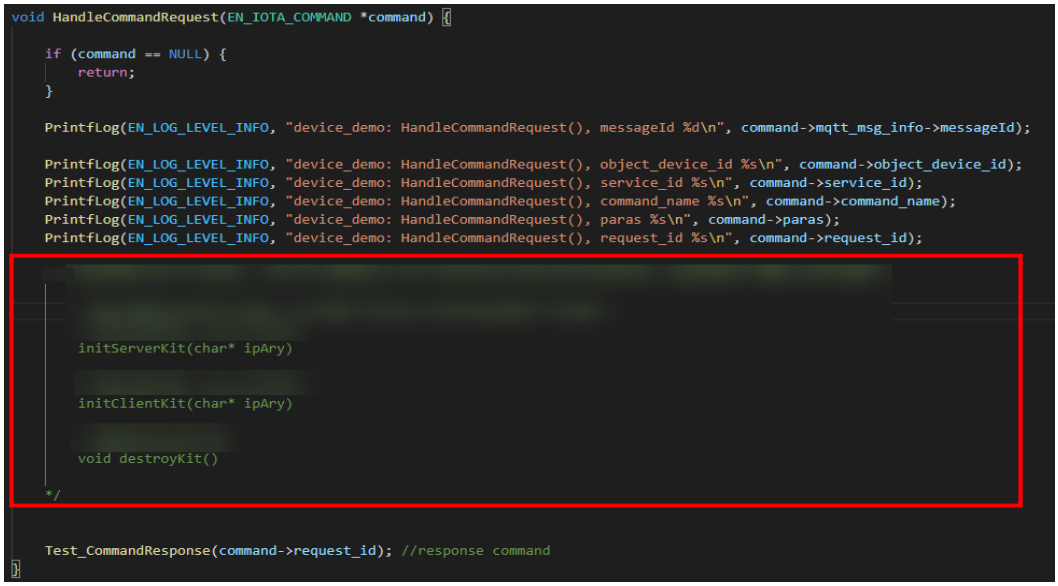
1. The platform delivers the soft bus version number. The device calls the API for obtaining the latest soft bus information to obtain and update the latest soft bus information.

2. If a device is added to the HarmonyOS group or the device IP address changes, the status of the HarmonyOS soft bus group is **To be synchronized**. When all soft bus information is synchronized to devices, the status is **Synchronized**.

Step 9 The device SDK calls an API to obtain the latest soft bus information and provides the obtained information for the HarmonyOS soft bus.

Step 10 When a device receives a downstream command or message that triggers a connection rule, the SDK calls the HarmonyOS soft bus API to implement device linkage. (The following figure uses command delivery in **device_demo.c** as an example.)

Figure 2-19 Example



----End

2.3 Rules

2.3.1 Overview

You can set rules for devices connected to IoTDA. If the conditions set in a rule are met, the platform triggers the corresponding action. Device linkage and data forwarding rules are available.

- **Data forwarding**

  Data forwarding is seamlessly interconnected with other services to implement full-stack services for device data storage, computing, and analysis.
- **Device linkage**

  When specific conditions are met, the platform triggers collaborative response of multiple devices to implement device linkage and intelligent control.

2.3.2 Data Forwarding

Overview

Data forwarding is seamlessly interconnected with other services to implement full-stack services for device data storage, computing, and analysis.

IoTDA can forward data to the following services/targets:

- Distributed Message Service (DMS) for Kafka: a message queuing service provided for data transfer. Kafka is distributed messaging middleware that features high throughput, data persistence, horizontal scalability, and stream data processing. It adopts the publish-subscribe pattern and is widely used for log collection, data streaming, online/offline system analytics, and real-time

monitoring. You can apply for instances as required and customize partitions and replicas for topics in the instances. The instances can be used right out of the box, taking off the deployment and O&M workload for you so that you can focus on developing your services.

- ROMA Connect: IoTDA can interconnect with ROMA Connect. It is a Message Queue Service (MQS) component that provides secure and standard message channels between IoTDA and applications. MQS is enterprise-level messaging middleware that uses Kafka and a unified message access mechanism. It provides basic and advanced functions to offer a unified message channel for enterprise data management. The basic functions include message publishing and subscription, topic management, user permissions management, resource statistics, monitoring, and alarm reporting. The advanced functions include message tracking, network isolation, and integration of cloud and on-premises applications.
- Third-party message queue (Kafka): Kafka server provided by a third party for data transfer. Kafka is distributed messaging middleware that features high throughput, data persistence, horizontal scalability, and stream data processing. It adopts the publish-subscribe pattern and is widely used for log collection, data streaming, online/offline system analytics, and real-time monitoring.
- Third-party application (HTTP push): You can call IoTDA APIs for creating a rule triggering condition, creating a rule action, and modifying a rule triggering condition or use the IoTDA console to configure and activate rules. After you specify an application URL, IoTDA pushes the changes to the specified URL server based on the type of data subscribed.
- MRS Kafka: IoTDA forwards device data to MRS Kafka for big data analytics and storage, facilitating flexible and diversified data use.
- OBS: IoTDA forwards device data to OBS for persistent storage. Currently, only JSON and CSV files are supported.
- InfluxDB: Device data is forwarded to the InfluxDB time series database through the IoTDA service for persistent storage. Enable SSL for the InfluxDB database.
- Vastbase G100: Device data can be forwarded to Vastbase G100 through IoTDA for persistent storage.
- Apache IoTDB: Data of devices is forwarded to Apache IoTDB for persistent storage.

For details, see **Subscription/Push**.

Procedure

Step 1 Log in to the IoTDA console.

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

Step 3 Set forwarding data.

Set parameters based on **Table 2-6** and click **Create Rule**. In the displayed dialog box, click **Continue**.

Figure 2-20 Setting forwarding data

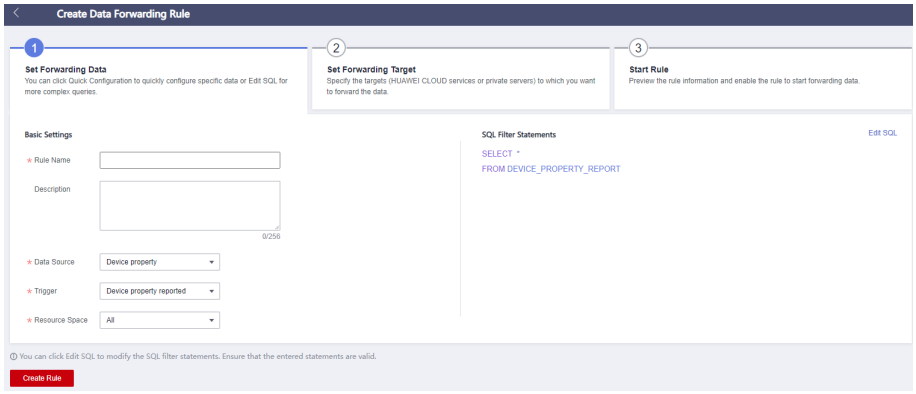


Table 2-6 Setting forwarding data

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

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

Parameter	Description
Edit SQL	You can edit the SQL statements for processing message data and set the data filtering statements. Click **Edit SQL** to edit the SQL statements for processing message fields. For details, see **SQL Statements**.

Step 4 Set the forwarding target.

Click **Add**, select a forwarding target by referring to **Table 2-7**, and click **OK**.

Figure 2-21 Add a forwarding target

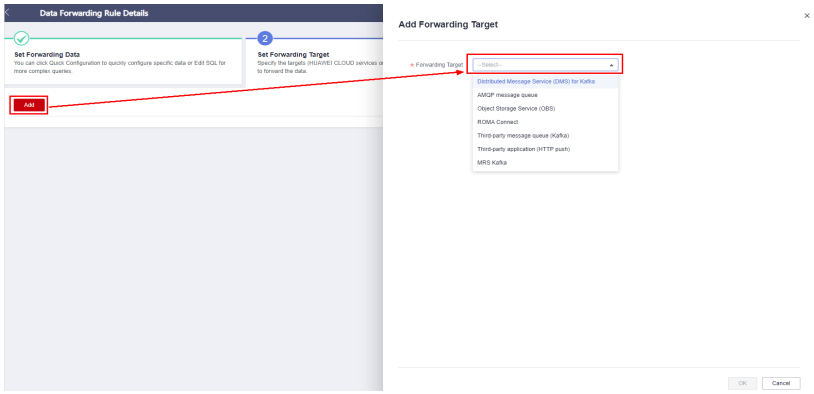


Table 2-7 Setting the forwarding target

Parameter	Description
Distributed Message Service (DMS) for Kafka **NOTE** Data can be forwarded only to Kafka premium instances.	• **Connection Address**:Specify the Kafka connection address. • **Topic**: Specify the topic of the forwarding target. • **Data Transmission Encryption**: If data transmission encryption is enabled, enter the SASL username and password that you entered when applying for a Kafka instance.
AMQP message queue	**Message Queue**: Select the queue to which messages are to be pushed. If no queue is available, create one.

Parameter	Description
ROMA Connect	ROMA Connect is focused on application and data integration. It integrates messages, data, APIs, and devices to help enterprises rapidly streamline legacy systems and cloud native applications. • **Connection Address**: Enter the connection address of MQS. On the **Instance Information** page of the ROMA Connect console, click the **Basic Information** tab to view the MQS public address in the connection address. • **Username/Password**: Enter the MQS username and password for logging in to the ROMA console. On the **Integration Applications** page of the ROMA Connect console, click the name of the integration application to which the topic belongs. In the **Basic Information** area of the **Overview** tab page, you can view the values of **Key** and **Secret**, that is, the username and password. • **Topic**: Specify the topic of the forwarding target. On the ROMA Connect console, choose **Message Queue Service** > **Topic Management** and view the topic name.
Third-party message queue (Kafka)	• **Connection Address**: Specify the connection address list of the Kafka server. • **Topic**: Specify the topic of the forwarding target. • **User Authentication Type**: The default type is **PAAS**, in which the Kafka server does not authenticate users. The Kafka server also supports PLAIN and SCRAM SASL authentication. Both modes are based on username and password. SCRAM is more secure than PLAIN and includes the SCRAM-SHA-256 and SCRAM-SHA-512 algorithms. • **Username/Password**: The SASL authentication is used. You need to enter the corresponding username and password.
Third-party application (HTTP push)	IoTDA can push specified device data to a third-party application based on the rule configured. You can set different addresses that different types of device data will be pushed to. For example, if the push URL is **https://10.10.10.10:8443/example/**.

Parameter	Description
MRS Kafka	MRS Kafka streaming clusters feature efficient stream data ingestion and real-time data processing and storage, meeting requirements of different big data application scenarios. They convert data into data models that meet service requirements based on the structure and logic. • **Kerberos Authentication**: Kerberos is a network authentication protocol used to authenticate client/server applications using the key encryption technology. To enable Kerberos authentication, configure the **krbFile** and **keytabFile** credential files. • **Connection Address**: Enter the connection address of MRS. • **Topic**: Customize a topic.

Parameter	Description
Object Storage Service (OBS) **NOTE** Currently, data can be forwarded only to OBS.	OBS is a stable, secure, cloud storage service that is scalable, efficient and easy-to-use. It allows you to store any amount of unstructured data in any format, and provides RESTful APIs so you can access your data from anywhere. • **Bucket Name**: Enter the name of the bucket created in OBS. If no OBS bucket is available, create one on the OBS console. • **Access Key ID (AK)** and **Secret Access Key (SK)** together make an access key. You can use an access key to sign requests of Cloud service APIs. AK is used together with SK to sign requests cryptographically, ensuring that the requests are secret, complete, and correct. For security purpose, SK is not displayed when you view action details. • **Endpoint**: OBS provides an endpoint for each region. An endpoint can be considered as the domain address of OBS in a region, and is used to process access requests from the region. For example, *{region}* or **12.12.12.12**. • **Custom Directory**: used to store files that will be dumped to OBS. Separate different directory levels are by slashes (/). The directory cannot start or end with a slash (/) or contain two or more consecutive slashes (/). • File Name: indicates the target file for data forwarding. If the file name is not specified, a Normal object is generated in OBS. If the file name is specified, an appendable object is generated in OBS. The normal object name cannot be used as the file name. • **File Type**: **JSON** and **CSV** are available. If you select **CSV**, select or enter existing fields in the JSON file and set target fields for storage.

Parameter	Description
	NOTE 1. The content size of each field in the CSV file cannot be greater than 32 KB. If greater, only the first 32 KB is used. 2. The content of the exported CSV file is UTF-8 encoded. 3. File name not specified: If the trigger source is not related to devices, the generated OBS file is named as follows: *Timestamp* + *Four-digit random number*. If the trigger source is related to devices, the OBS file is named as follows: *Device ID* + *Timestamp* + *Four-digit random number*. In a project, **you cannot set the same file in the same OBS bucket as the forwarding destination for multiple rules.** 4. If the endpoint is set to a domain name, **DNS resolution may fail.** Contact technical support engineers.
Time series database InfluxDB	• **Database Instance Address**: Enter the address of the InfluxDB database. • **Database Name**: indicates the target database in InfluxDB. If the target database does not exist, it is automatically created and kept for 90 days by default. • **Access Account**: Enter the account with access permissions. • **Access Password**: Enter the password of the account. • **Save to**: Enter the target table where data is to be stored. If the table does not exist, it will be automatically created. • **Forwarding Field**: Enter the data field to be forwarded. • **Target Field**: Enter the target field in the database table. **NOTE** 1. If multiple actions of forwarding data to InfluxDB use the same database and table and set the same target fields for forwarding different types of data, the forwarding fails. 2. Do not set the **Target Field** to **time**, which may conflict with the default field of InfluxDB.

Parameter	Description
Vastbase G100	Vastbase G100 is an enterprise-level relational database developed by Vastdata based on the openGauss kernel. • **Database Instance Address**: Enter the address of Vastbase G100. • **Database Name**: indicates the target database in Vastbase G100. • **Access Account**: Enter the account with access permissions. • **Access Password**: Enter the password of the account. • **SSL**: SSL is enabled by default. • **Encryption Certificate**: Select the certificate uploaded to the platform when SSL is enabled. • **Save to**: Enter the target table where data is to be stored. **Use lowercase letters to name tables created on the database server.** By default, the table name returned on the console is in uppercase letters. • **Forwarding Field**: Enter the data field to be forwarded. • **Target Field**: Enter the target field in the database table.

Parameter	Description
Time series database (Apache IoTDB)	Apache IoTDB is a low-cost and high-performance IoT-native time series database. • Database URL: address of the IoTDB database. REST must be enabled for IoTDB (**enable_rest_service=true**). • Access account: account with required permissions, which must have the **READ_DATA**, **WRITE_DATA**, **READ_SCHEMA**, and **WRITE_SCHEMA** permissions on the corresponding path. • Access password: Enter the password of the account. • Dump to path: IoTDB data storage path. Set the expiration time of the IoTDB database path to prevent the disk from being used up. • SSL: SSL is enabled by default. • Certificate file: If SSL is enabled, select the certificate that has been uploaded to the platform. • Target dump field: target field in the database. **NOTE** 1. If the same path is set for multiple IoTDB forwarding actions and the target storage fields are the same, the target storage field value may be null if the data types of the forwarding fields of these forwarding actions are different. 2. The target storage field cannot be set to time, which will conflict with the default field Time of IoTDB.

Step 5 Enable a rule.

After the rule is configured, click **Enable Rule** to start data forwarding.

----End

2.3.3 SQL Statements

When creating a data forwarding rule, you must compile SQL statements to parse and process JSON data reported by devices. Data in binary format is transparently transmitted without being parsed. This topic describes how to compile SQL statements used in data forwarding rules.

SQL Statements

An SQL statement consists of the SELECT and WHERE clauses. Each clause can contain a maximum of 500 characters. Chinese and other characters are not supported. Contents in the SELECT and WHERE clauses are case-sensitive. However, keywords such as SELECT, WHERE, and AS are case-insensitive.

The following provides an example of SQL statements used in a data forwarding rule.

The source data is as follows:

```
{
  "resource" : "device.message",
  "event" : "report",
  "event_time" : "20151212T121212Z",
  "notify_data" : {
    "header" : {
      "device_id" : "d4922d8a-6c8e-4396-852c-164aefa6638f",
      "product_id" : "ABC123456789",
      "app_id" : "d4922d8a-6c8e-4396-852c-164aefa6638f",
      "gateway_id" : "d4922d8a-6c8e-4396-852c-164aefa6638f",
      "node_id" : "ABC123456789",
      "tags" : [ {
        "tag_value" : "testTagValue",
        "tag_key" : "testTagName"
      } ]
    },
    "body" : {
      "topic" : "topic",
      "content" : {
          "temperature" : 40,
          "humidity" : 24
      }
    }
  }
}
```

To trigger the rule when the temperature is greater than 38°C and filter out the device name, temperature, and humidity, use the following SQL statement:

```
SELECT notify_data.body.content.temperature AS t, notify_data.body.content.humidity AS humidity,
notify_data.header.device_id AS device_id
WHERE notify_data.body.content.temperature > 38
```

The format of the forwarded data is as follows:

```
{
   "t": 40,
   "device_id": "d4922d8a-6c8e-4396-852c-164aefa6638f",
   "humidity": 24
}
```

SELECT Clause

The SELECT clause consists of **SELECT** followed by multiple SELECT subexpressions, which can be *, JSON variables, string constants, or integer constants. A JSON variable is followed by an AS keyword and an AS variable, 32 characters in total. If a constant is used, you must use AS to specify the name.

- JSON variable

  A JSON variable can contain letters, numbers, underscores (_), and hyphens (-). To distinguish a hyphen (-) from the minus sign, use double quotation marks to enclose the JSON variable with a hyphen, for example, **$."msg-type"**.

  $. can be omitted to make the statement more concise.

  The JSON variable extracts data of the nested structure.

  ```
  {
    "a":"b",
    "c": {
  ```



```
  "d" : "e"
 }
}
```

$.c.d can be used to extract character string **e**, which can be nested at multiple layers.

- AS variable

  An AS variable consists of letters and is case sensitive. The variable **[a-zA-Z_-]*** is supported. If a hyphen (-) is used, enclose it with double quotation marks ("").

WHERE Clause

In the WHERE clause, you can perform Boolean operations using JSON variables, make some non-null judgments, and combine the results using AND or OR.

- **IS NULL and IS NOT NULL**

  Null judgment can be used in the WHERE clause. If the JSON variable cannot extract data or the extracted array is empty, **IS NULL** is true. Otherwise, **IS NOT NULL** is true.

  ```
  WHERE $.data IS NULL
  ```

- **Operators > <**

  The greater than (>) or less than (<) operator can be used in the WHERE clause. The operator can be used between two JSON variables, between a JSON variable and a constant, or between a constant and a constant only when the value of a JSON variable is a constant integer.

- **Equals sign (=)**

  The equals sign (=) can be used in the WHERE clause for comparison between JSON variables, between JSON variable integers and integer constants, and between JSON variable strings and string constants. If **IS NULL** for the two JSON variables is true, the comparison result of the equals sign (=) is false.

- **IN/NOT IN**

  IN and **NOT IN** operators are used in **WHERE** clauses. If the target is in the specified value set, **IN** is valid. Otherwise, **NOT IN** is valid. **IN** and **NOT IN** operators support only strings and digits. An **IN** or **NOT IN** set supports only constants. The element types in the set must be the same as those of the target values.

  Example:

  ```
  notify_data.header.product_id IN ('product_id1', 'product_id2')
  ```

Function List

Multiple functions are used in rules. You can use these functions when compiling SQL statements to implement diversified data processing. Function names must be capitalized.

```
SELECT GET_TAG('tagA') AS tag FROM DEVICE_PROPERTY_REPORT
```


Function Name	Parameter	Purpose	Return Value Type	Restrictions
GET_TAG	String tagKey	Obtain the value of a tag.	String	-
CONTAINS_TAG	String tagKey	Determine whether a tag is contained.	Boolean	-
GET_SERVICE	String serviceId	Obtain the service with a specific **serviceId**. If you have multiple services with the same **serviceId** in a message body, the result may be unreliable.	JSON structure	Used only for property reporting
GET_SERVICES	String serviceId	Obtain the service with a specific **serviceId**. If you have multiple services with the same **serviceId** in a message body, the result may be unreliable.	JSON array	Used only for property reporting
CONTAINS_SERVICES	String serviceId	Obtain the service with a specific **serviceId** and combine the results into an array.	Boolean	Used only for property reporting
GET_SERVICE_PROPERTIES	String serviceId	Obtain the properties field in the service with a specific **serviceId**.	JSON structure	Used only for property reporting
GET_SERVICE_PROPERTY	String serviceId, String propertyKey	Obtain the value of **propertyKey** in **properties** of a service with a specific **serviceId**.	String	Used only for property reporting

STARTS_WITH	String input, String prefix	Check whether the input value starts with **prefix**.
		Example:
		`STARTS_WITH('abcd','abc')`
		`STARTS_WITH(notify_data.header.device_id,'abc')`
		`STARTS_WITH(notify_data.header.device_id,notify_data.header.product_id)`
			Boolean	-

Function Name	Parameter	Purpose	Return Value Type	Restrictions

ENDS_WITH	String input, String suffix	Check whether the input value ends with **suffix**.
		Example:
		`ENDS_WITH('abcd','bcd')`
		`ENDS_WITH(notify_data.header.device_id,'abc')`
		`ENDS_WITH(notify_data.header.device_id,notify_data.header.node_id)`
			Boolean	-

CONCAT	String input1, String input2	Concatenate the **input1** and **input2** strings.
		Example:
		`CONCAT('ab','cd')`
		`CONCAT(notify_data.header.device_id,'abc')`
		`CONCAT(notify_data.header.product_id,notify_data.header.node_id)`
			String	-

REPLACE	String input, String target, String replacement	Replace **target** (part of the string) in the input value with **replacement**. Example: `REPLACE(notify_data.header.node_id,'nodeId','IMEI')`	String	-

SUBSTRING	String input, int beginIndex, int endIndex(required=false)	Obtain the substring that is returned and ranges from **beginIndex** (included) to **endIndex** (excluded) of the input value.
		Note: **endIndex** is optional.
		`SUBSTRING(notify_data.header.device_id,3)`
		`SUBSTRING(notify_data.header.device_id,3,12)`
			String	-

NOW	-	Obtain the timestamp of the current system. The unit is ms.	Integer	-

Function Name	Parameter	Purpose	Return Value Type	Restrictions
GET_SERVICES_BY_TIME_QUERY	long startTime, long endTime	Obtain services that meet the conditions of **startTime < event_time < endTime** from **services** and return a service list. **startTime** and **endTime** are timestamps. `GET_SERVICES_BY_TIME_QUERY(0, 1000000)`	JSON array	Used only for property reporting

LENGTH	String jsonPath/ Functions	Calculate the length of the array obtained by **jsonPath** or functions.
		`LENGTH(GET_SERVICES_BY_TIME_QUERY(0, 1000000))`
		`LENGTH(notify_data.body.services)`
			Integer	The result of the input parameter must be an array.

FILTER_SERVICES_BY_TIME_QUERY	long startTime, long endTime	Filter services that meet the conditions of **startTime < event_time < endTime** and return the JSON body containing the filtered service list. **startTime** and **endTime** are timestamps. `FILTER_SERVICES_BY_TIME_QUERY(0, 1000000)`	JSON structure	-

GET_CHILDREN	String jsonPath/ Functions	Obtain the JSON structure body from the jsonPath or the function.
		`GET_CHILDREN(notify_data.body)`
		`GET_CHILDREN(FILTER_SERVICES_BY_TIME_QUERY(0, 1000000))`
			JSON structure	1. The result of the input parameter must be in the format of JSON structure body.
				2. This function cannot be used with AS.

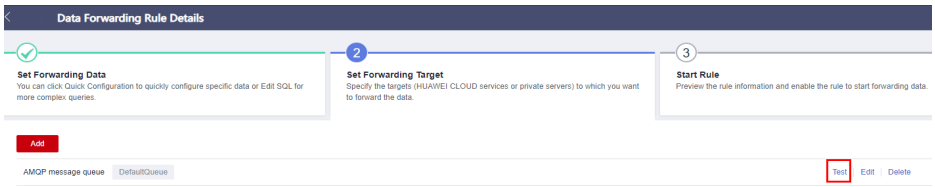
2.3.4 Connectivity Tests

Overview

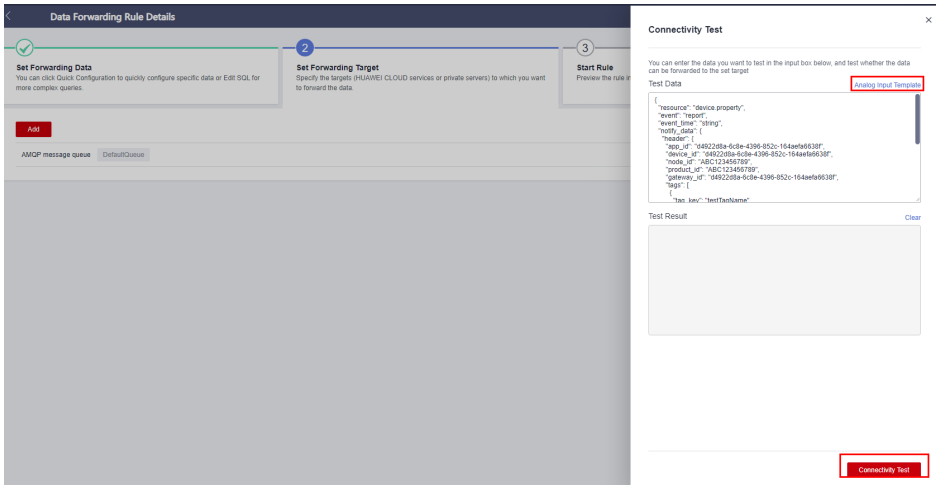
IoTDA provides connectivity tests on the forwarding targets. In the service debugging phase, you can simulate service data to test the availability of rule actions and the consistency of forwarded data. If a fault occurs in data forwarding during service running, you can perform connectivity tests to reproduce and locate the fault.

Procedure

Step 1 After creating a forwarding rule, click **Test** in row of the forwarding target to be debugged.



Step 2 In the **Connectivity Test** dialog box, enter the test data for forwarding in **Test Data**, or click **Analog Input Template** in the upper right corner to use the template data, and then click **Connectivity Test**.



----End

2.3.5 Server Certificates

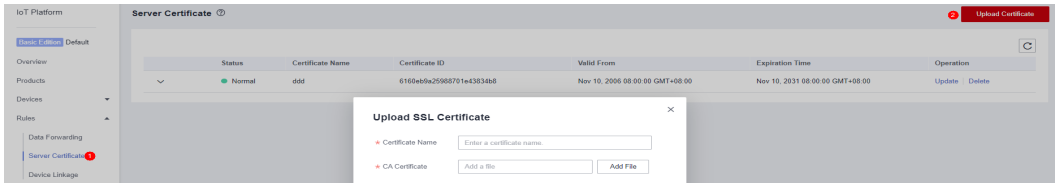
IoTDA supports interactions with third-party applications. To ensure transmission security between the platform and applications during access, SSL certificate verification is required. You can centrally manage SSL certificates on the **Server Certificate** page to use the same SSL certificate in the same service scenario.

Uploading an SSL Certificate

This section describes how to upload a certificate using HTTPS.

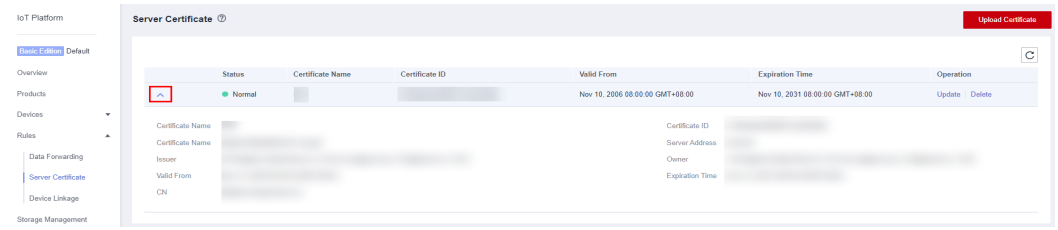
Step 1 Log in to the IoTDA console.

Step 2 In the navigation pane, choose **Rules** > **Server Certificate**. Click **Upload Certificate** in the upper right corner, configure parameters based on the following table, and click **OK**.



Parameter	Description
Certificate Name	It is used to distinguish different certificates and can be customized.
CA Certificate	A CA certificate from the application can be applied for in advance. **NOTE** You can **create a commissioning certificate** during commissioning. For security reasons, you are advised to replace the commissioning certificate with a commercial certificate during commercial use.

Step 3 In the navigation pane, choose **Rules** > **Server Certificate** and locate the target certificate to obtain the certificate ID, which is used as a parameter in the API for creating a rule action.



----End

Managing SSL Certificates

After the certificate is uploaded, you can view and manage the certificate in the server certificate list.

- Updating a certificate: In the certificate list, locate the target certificate and click **Update** to upload a new certificate.
- Deleting a certificate: Locate the certificate to be deleted and click **Delete**. In the displayed dialog box, select **Agree to delete**, and click **OK** to delete the certificate. After the certificate is deleted, authentication functions based the certificate will be affected. For example, the message push through HTTPS will be affected.

2.3.6 Device Linkage

When specific conditions are met, the platform triggers collaborative response of multiple devices to implement device linkage and intelligent control.

2.3.6.1 Cloud Rules

Overview

If you set a cloud rule, IoTDA determines whether the rule triggering condition is met. If the condition is met, IoTDA performs actions you set, such as alarm reporting, topic notification, and command delivery.

Procedure

Step 1 Log in to the IoTDA console.

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

Step 3 Create a device linkage rule based on the table below.

Parameter	Description
Rule Name	Name of a rule to be created.
Activate upon creation	You can set whether to activate the rule upon creation.
Rule Type	Set the **Rule Type** to **Cloud**.
Effective Period	• **Always effective**: There is no time limit. The platform always checks whether the conditions are met. • **Specific time**: You can select a time segment during which the platform checks whether the conditions are met.
Description	Description of the rule.

Parameter	Description
Condition	You can set whether all conditions or any of the conditions need to meet. • **Triggered upon matching device**: Set conditions for devices that use the same product model. – **Select product**: Select a specific product. – **Select service**: Select a service type. – **Select property**: Select a property in the data reported. **NOTE** ▪ If the data type of a property is **int**, **long**, or **decimal**, you can select multiple operators. ▪ When the data type of a property is **string**, **date time**, **jsonObject**, **boolean**, **enum**, or **string list**, you can only select the equal sign (=) as the operator. – Property comparison value: Enter values and an expression. The expression supports only asset properties. If an asset property is selected, the device property value in the reported data is compared with the selected asset property value to determine whether the condition is met. Configure asset properties of devices by referring to **Asset Properties**. – **Data Validity Period (s)**: Specify the data validity period. For example, if **Data Validity Period** is set to 30 minutes, a device generates data at 19:00, and the platform receives the data at 20:00, the action is not triggered even if the conditions are met. • **Triggered upon specified device**: Set conditions for a specified device. Select the device, service, and property as the conditions. For details, see the parameters described under **Triggered upon matching device**. • **Optional: Triggered periodically**: Set the time at which the rule is triggered. It is usually used for periodic triggering conditions, such as turning off street lights at 07:00 every day. **NOTE** If **Timer** is selected, **Send notifications**, **Report alarms**, and **Clear alarms** cannot be selected for **Actions**. – By period: Select a week or a time point. – By policy: ▪ Specify a start time for triggering the rule. ▪ **Repeat**: Enter the number of times that the rule can be repeatedly triggered. The value ranges from 1 to 9999. ▪ **Interval**: interval for triggering the rule after the start time. The value ranges from 1 to 525600, in units of minutes.

Parameter	Description
Set Actions	Click **Add Action** to set the action to execute after the rule is triggered. • **Deliver commands**: Select the device, service, and command to be delivered in sequence, and set the command delivery parameters. • **Send notifications**: Select the region where the SMN service is located. If the platform has not been granted with the permissions to access SMN, perform the authorization as prompted. Click the corresponding link to go to the SMN console and set the topic. • **Report alarms**: Define the alarm severity, name, and content. • **Clear alarms**: Define the alarm severity and name. If the conditions are met, the alarm reported by the device to the platform is cleared.

Step 4 Click **Create Rule** in the lower right corner. Newly created rules are in the activated state by default. You can disable a rule in the **Status** column of the rule list.

----End

2.3.6.2 Device-side Rules

Overview

Cloud rules are parsed and executed on the cloud. IoTDA determines whether triggering conditions are met and triggers corresponding device linkage actions. Device-side rules are device linkage rules delivered to devices, where the device-side rule engine parses and executes the rules. Device-side rules can still run on devices when the network is interrupted or devices cannot communicate with the platform. Device-side rules can extend application scenarios and improve device running stability and execution efficiency. For example, when the indoor light intensity is lower than 20 lx, lights can be automatically turned on. This implements intelligent control independent of network devices.

Typical scenarios

There are a large number of surveillance devices in highway tunnels. The network environment is complex and the network quality is unstable. However, emergency handling has high requirements on real-time network performance. Linkage between emergency devices cannot completely depend on cloud rules. Device-side rules are required to implement emergency plan linkage. Device linkage plans can be formulated in advance based on different situations such as fires and traffic accidents. Monitoring personnel can start device linkage plans with one click based on tunnel conditions. Device-side rules enable simultaneous status changes of different types of devices. This reduces dependency on network quality and improves overall device linkage efficiency. For example, if the temperature of a flue pipe is too high, the controller can be linked to open the drainage valve to

reduce the temperature. If the concentration of carbon monoxide (CO) is too high, a COVI device can be linked to control fans for ventilation.

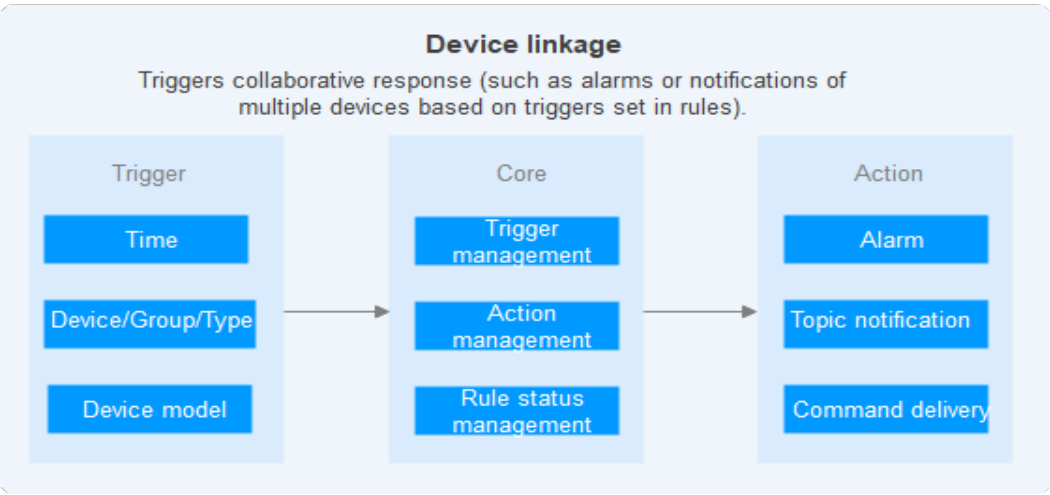
Prerequisites

1. Device-side rules support only command delivery actions.

2. Devices must be integrated with IoT Device SDK (C) v1.1.2 or later.

3. Devices need to report the SDK version number to IoTDA using the APIs provided by the SDK.

Figure 2-22 Device-side rule architecture



Example

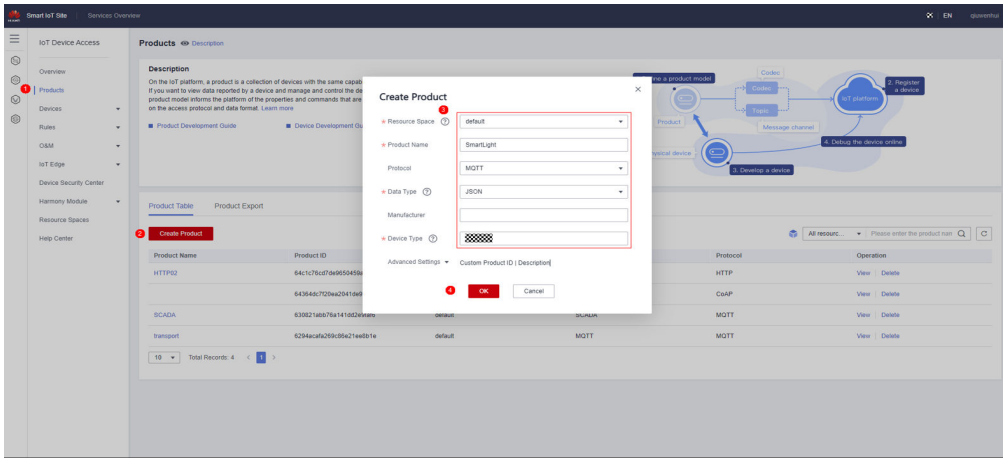
The following uses a smart street light system as an example to describe how to use device-side rules.

Step 1 Log in to the IoTDA console.

Step 2 Create a product and model.

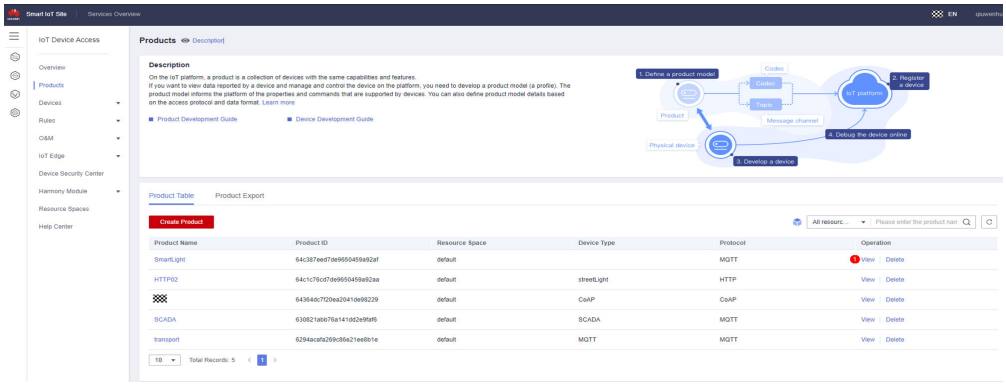
1. To create a product, choose **Products** > **Create Product**, enter the product name and choose the device type, and click **OK**.

Figure 2-23 Creating a product



2. A model is created. Find **SmartLight** in the product list and click **View**.

Figure 2-24 Viewing product information



3. On the product details page, click **Customize Model** and add two services: BasicData and LightControl, as shown in the following figures.

Figure 2-25 Customizing model: service BasicData

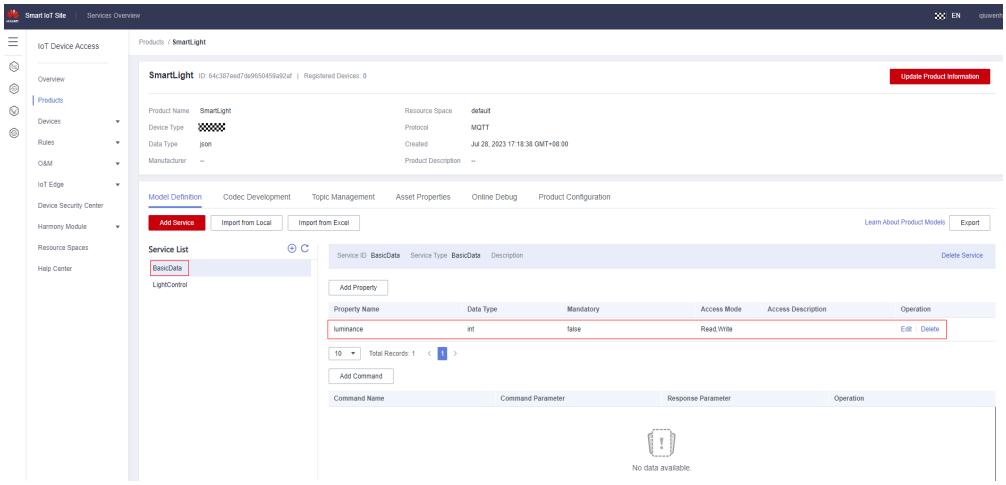
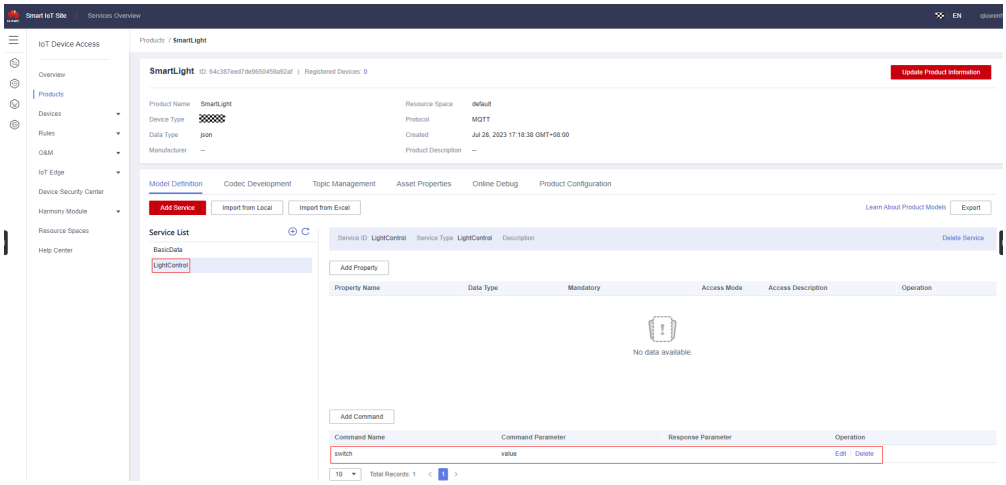
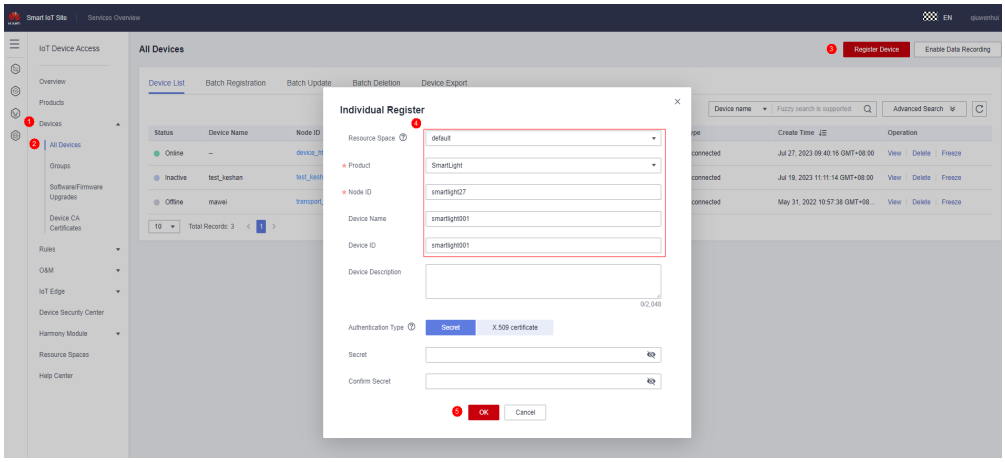


Figure 2-26 Customizing model: service LightControl



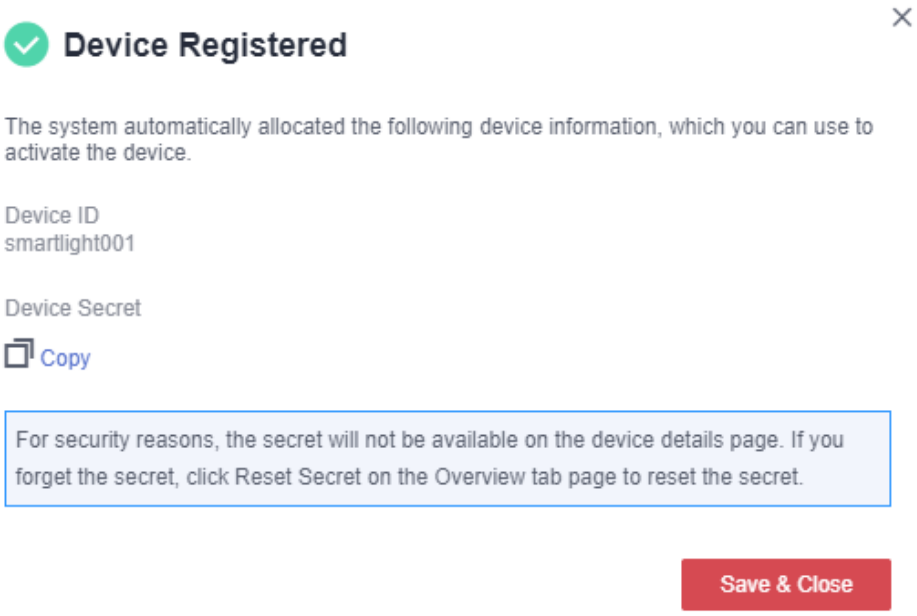
Step 3 In the navigation pane, choose **Devices** > **All Devices** and click **Individual Register** in the upper right corner. Select the resource space you select in **Step 2** and a product, enter a node ID, and click **OK**.

Figure 2-27 Creating a device



Step 4 After the device is created, copy and save the device secret for later use.

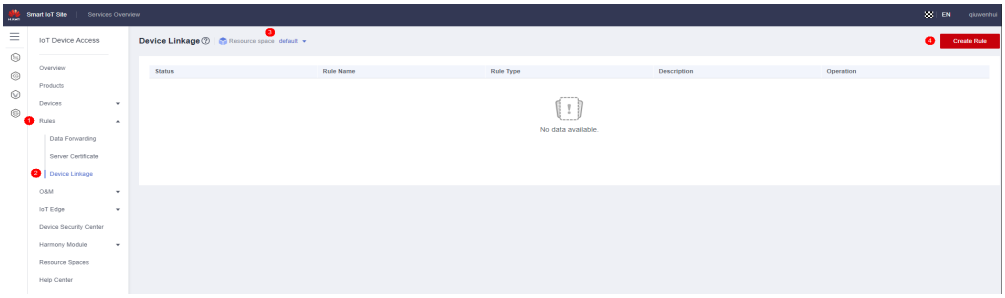
Figure 2-28 Device registered



Step 5 Create a rule.

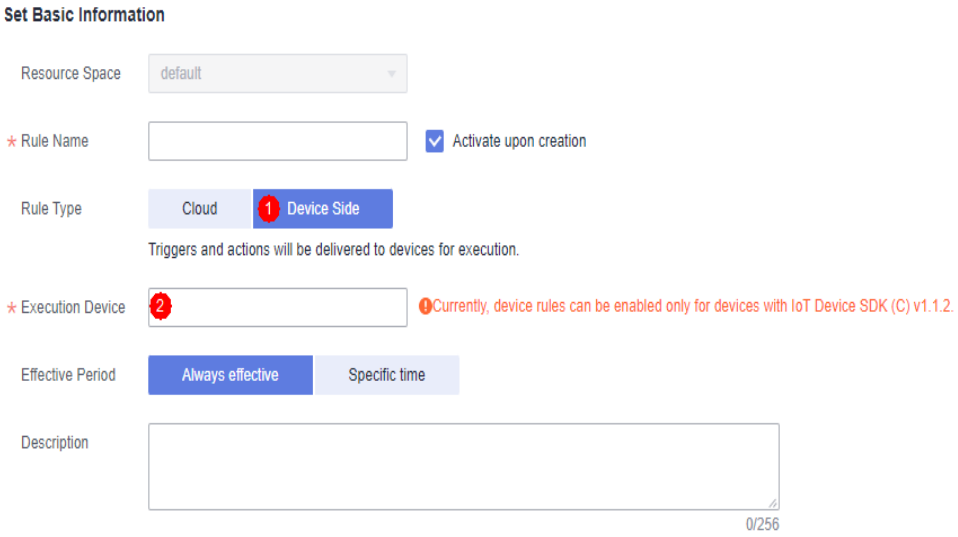
1. In the navigation pane, choose **Rules** > **Device Linkage**. In the upper part of the page, select the resource space of the product to which the device belongs. Click **Create Rule** in the upper right corner.

Figure 2-29 Creating a rule



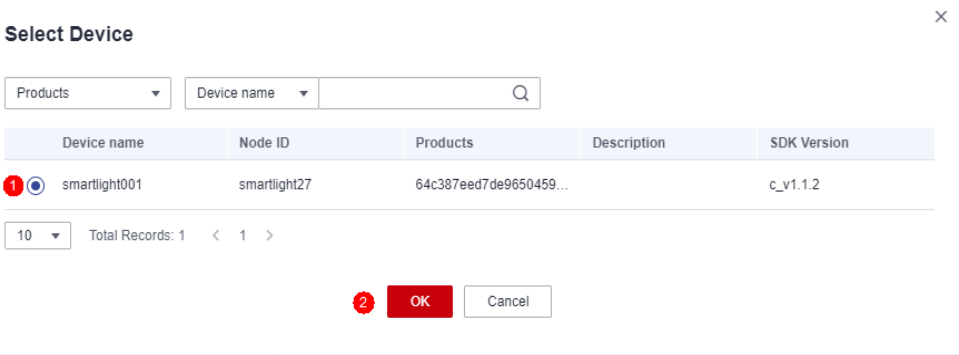
2. On the page for creating a rule, enter a rule name, select **Device Side** for **Rule Type**, and select a device for **Execution Device**. The rule will be delivered to the device you select for parsing and execution.

Figure 2-30 Selecting Device Side



3. Select **smartlight001** and click **OK**.

Figure 2-31 Selecting a rule execution device



NOTE

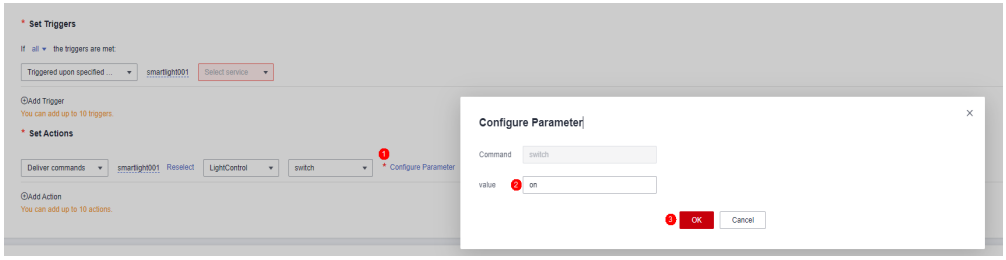
Device-side rules can be created only for devices with the IoT Device SDK. Currently, only IoT Device SDK (C) v1.1.2 is supported. IoTDA obtains on-device SDK version from information reported by southbound devices through the related API (Refer to the API reference).

4. Click **Add Trigger**. The current device is used by default, and other devices are not available. Click **Add Action** and select the current device or other devices.

Figure 2-32 Adding a trigger and action

5. In **Set Triggers**, set the property trigger to **luminance<= 27**. In **Set Actions**, configure the **control_light** command and configure the parameter to set **light_state** to **on**.

Figure 2-33 Setting a trigger condition and action



Step 6 Create a device linkage rule based on the table below.

Table 2-8 Parameter description

Parameter	Description
Rule Name	Name of a rule to be created.
Activate upon creation	Selected: The rule is activated upon creation. Deselected: The rule is not activated after creation.

Parameter	Description
Effective Period	• **Always effective**: There is no time limit. IoTDA always checks whether conditions are met. • **Specific time**: You can select a time segment during which the platform checks whether the conditions are met. **NOTE** Device-side rules are stored in the memory. When a device is powered off, rules stored on the device are cleared. When the device is restarted or powered on, the device updates all historical rules from IoTDA.
Description	Description of the rule.
Set Triggers	You can set whether all conditions or any of the conditions need to meet. **NOTE** If all conditions need to meet, **Triggered upon specified device** and **Triggered periodically** modes cannot be used at the same time. You can specify different devices to set multiple triggers based on the devices. Trigger type: Currently, only **Device Property**, **Device Status**, and **Timer** are supported. • **Triggered upon specified device**: The rule will be triggered when the specified device reports properties. – **Select service**: Select the corresponding service type. – **Select property**: Select a property in the data reported. **NOTE** – If the data type of a property is **int** or **decimal**, you can select multiple operators. – If the data type of a property is **string**, you can only select the equal sign (=) as the operator. • **Triggered periodically**: You can customize the trigger. – Specify a start time for triggering the rule. – **Repeat**: number of times that the rule can be triggered. The value ranges from 1 to 1440. – **Interval**: interval for triggering the rule after the start time. The value ranges from 1 to 1440, in units of minutes.
Set Actions	Click **Add Action** to set the action to execute after the rule is triggered. **Deliver commands**: Select the device, service, and command to be delivered in sequence, and set the command delivery parameters.

Step 7 Click **Create Rule** in the lower right corner. Newly created rules are in the activated state by default. You can disable a rule in the **Status** column of the rule list.

Step 8 Compile the device-side code. In SDKs that support device-side rules (only IoT Device SDK C is supported currently), you only need to implement the callback functions for property reporting and command processing. Click **here** to obtain the

IoT Device SDK (C) and perform the following operations after the operations in **Preparations** are complete.

1. Open the **src/device_demo/device_demo.c** file and find the **HandleCommandRequest** function.

   Figure 2-34 Command processing

   ```
   void HandleCommandRequest(EN_IOTA_COMMAND *command)
   {
       if (command == NULL) {
           return;
       }

       PrintfLog(EN_LOG_LEVEL_INFO, "device_demo: HandleCommandRequest(), messageId %d\n",
           command->mqtt_msg_info->messageId);

       PrintfLog(EN_LOG_LEVEL_INFO, "device_demo: HandleCommandRequest(), object_device_id %s\n",
           command->object_device_id);
       PrintfLog(EN_LOG_LEVEL_INFO, "device_demo: HandleCommandRequest(), service_id %s\n", command->service_id);
       PrintfLog(EN_LOG_LEVEL_INFO, "device_demo: HandleCommandRequest(), command_name %s\n", command->command_name);
       PrintfLog(EN_LOG_LEVEL_INFO, "device_demo: HandleCommandRequest(), paras %s\n", command->paras);
       PrintfLog(EN_LOG_LEVEL_INFO, "device_demo: HandleCommandRequest(), request_id %s\n", command->request_id);
                                                                   <-- Implement the command here.
       Test_CommandResponse(command->request_id); // response command
   }
   ```

   The following commands are use for demonstration only.

   ```
   printf("----- execute command----- \n");
   printf("service_id: %s\n", command->service_id);
   printf("command_name: %s\n", command->command_name);
   printf("paras: %s\n", command->paras);
   ```

2. Open the **src/device_demo/device_demo.c** file and find the **TestPropertiesReport** function.

   Figure 2-35 Replacing the code

   ```
   void Test_PropertiesReport()
   {
       const int serviceNum = 2; // reported services' totol count
       ST_IOTA_SERVICE_DATA_INFO services[serviceNum];

       //---------------the data of service1------------------------------
       char *service1 = "{\"Load\":\"6\",\"ImbA_strVal\":\"7\"}";

       services[0].event_time =
           GetEventTimesStamp(); // if event_time is set to NULL, the time will be the iot-platform's time.
       services[0].service_id = "parameter";
       services[0].properties = service1;

       //---------------the data of service2------------------------------   <-- Replace the code here.
       char *service2 = "{\"PhV_phsA\":\"9\",\"PhV_phsB\":\"8\"}";

       services[1].event_time = NULL;
       services[1].service_id = "analog";
       services[1].properties = service2;

       int messageId = IOTA_PropertiesReport(services, serviceNum, 0, NULL);
       if (messageId != 0) {
           PrintfLog(EN_LOG_LEVEL_ERROR, "device_demo: Test_PropertiesReport() failed, messageId %d\n", messageId);
       }

       MemFree(&services[0].event_time);
   }
   ```

   Use the following code:

   ```
   const int serviceNum = 1; // reported services' total count
   ST_IOTA_SERVICE_DATA_INFO services[serviceNum];
   ```



```
#define MAX_BUFFER_LEN 70
char propsBuffer[MAX_BUFFER_LEN];
// This is an example of obtaining a temperature value. Obtain the actual value from a sensor.
if(sprintf_s(propsBuffer, sizeof(propsBuffer), "{\"luminance\": %d}", 20) == -1){
printf("can't create string of properties\n");
return;
}

services[0].event_time = GetEventTimesStamp(); // if event_time is set to NULL, the time will be the
iot-platform's time.
services[0].service_id = "BasicData";
services[0].properties = propsBuffer;

int messageId = IOTA_PropertiesReport(services, serviceNum, 0, NULL);
if (messageId != 0) {
printf("report properties failed, messageId %d\n", messageId);
}
free(services[0].event_time);
```

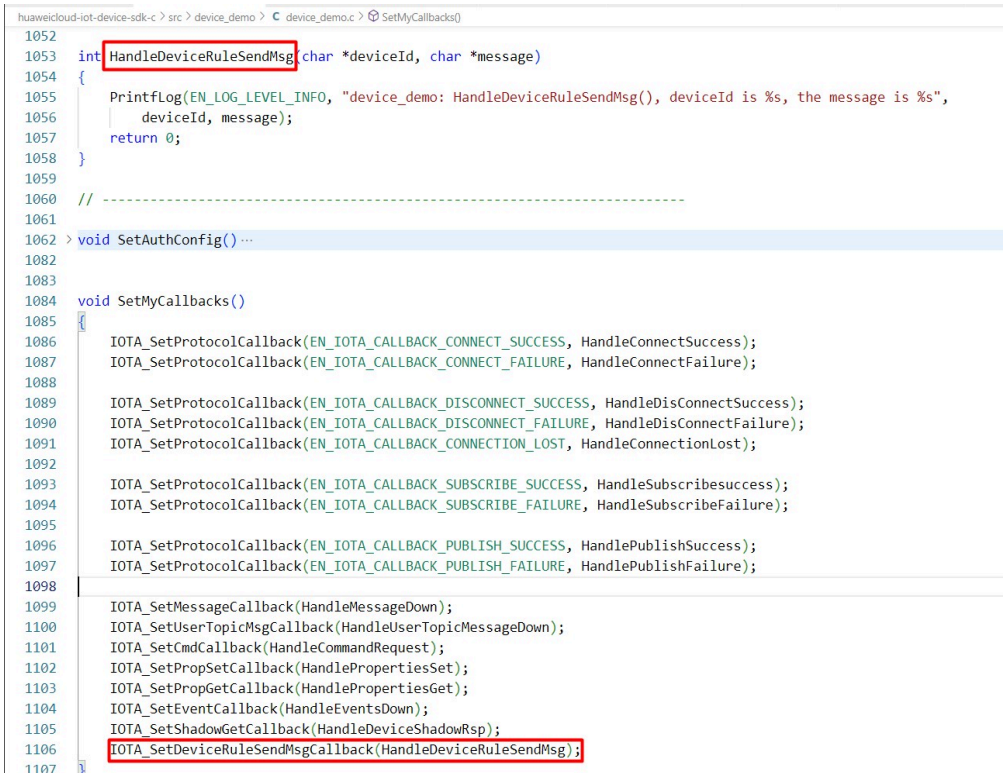
3. Compile and run the SDK. You can see the corresponding command from the output.

```
----- execute command-----
service_id: BasicData
command_name: control_light
paras: {
        "light_state":       "on"
}
```

The preceding log is only an example. You need to implement the specific command processing code in 1.

If a command is executed across devices, the callback function **IOTA_SetDeviceRuleSendMsgCallback** needs to be called. You need to implement message sending using the preconfigured function **HandleDeviceRuleSendMsg**. After the command execution device receives the message, it parses and executes the command.

Figure 2-36 Parsing and executing commands



----End

2.3.7 Data Forwarding Flow Control Policy

Overview

Data forwarding flow control allows you to configure flow control policies based on your service scenarios and forwarding target performance to protect your backend services.

Procedure

Step 1 Log in to the IoTDA console.

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

Step 3 On the flow control policy page, click **Create Policy**, set parameters by referring to the following table, and click **OK**.

Table 2-9 Flow control policy parameters

Parameter	Description
Policy Name	Name of the flow control policy.
Description	Function and purpose of creating the flow control policy.

Parameter	Description
Flow control size	Maximum TPS for data forwarding.
Policy Type	There are three types of flow control policies: **Forwarding channel**, **Forwarding rule**, and **Forwarding action**. • **Forwarding channel** indicates that the flow control policy is applied to a specified forwarding channel. Types of channels include: – Distributed Message Service (DMS) for Kafka – AMQP message queue – ROMA Connect – Third-party message queue (Kafka) – Third-party application (HTTP push) – MRS Kafka – OBS – InfluxDB – Vastbase G100 – Apache IoTDB • **Forwarding rule** indicates that the flow control policy is applied to a specified forwarding rule. • **Forwarding action** indicates that the flow control policy is applied to a specified forwarding action.

Step 4 On the policy configuration page, you can view, modify, and delete created flow control policies. You can only modify the name, description, and flow control value of a policy.

----End

2.4 Monitoring and O&M

2.4.1 Reports

IoTDA provides a variety of reports to intuitively display data. You can view statistics of different resource spaces. The statistics include device statuses, device messages, total number of API calls, number of different response codes, number of forwarded messages, and data forwarding traffic.

In the left navigation pane, choose **O&M** > **Reports**. On the **Reports** page that is displayed, you can view the statistics details. **Table 2-10** describes the report names and functions.

Figure 2-37 Device monitoring statistics

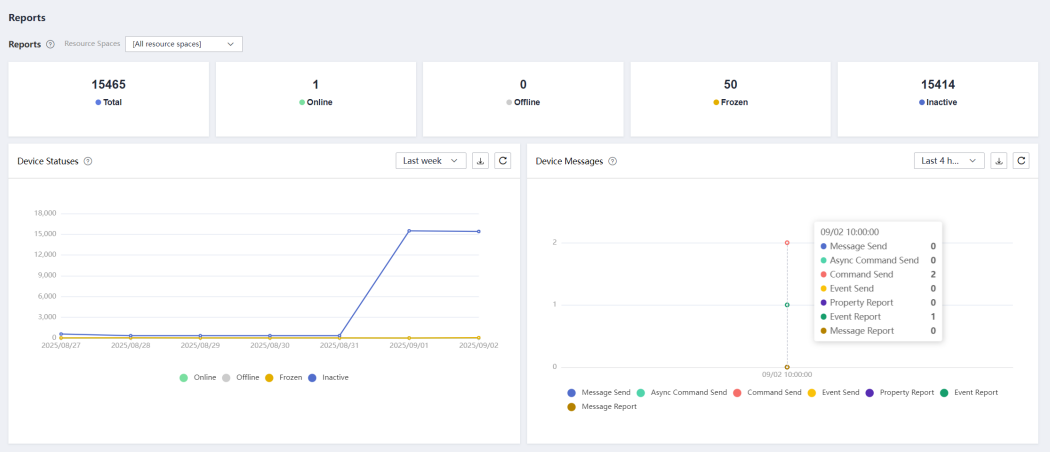
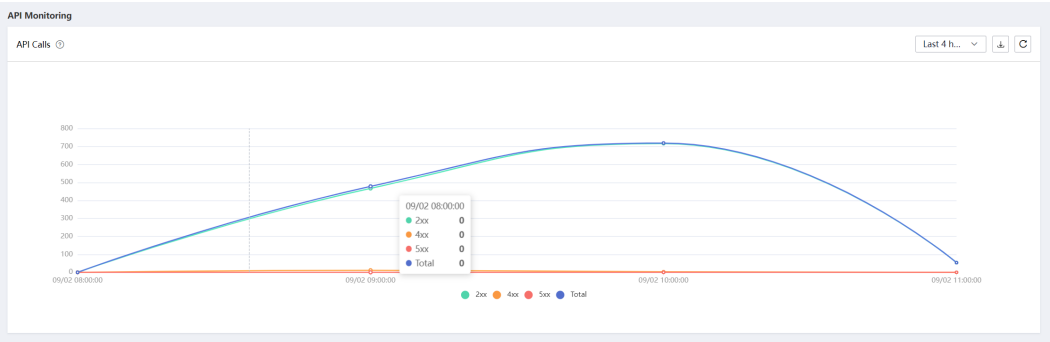


Figure 2-38 API calls



NOTE

The rules for viewing API call statistics are as follows:

- If the login account has the admin permission, you can view all statistics of the tenant. Otherwise, you can view only the statistics of the specific account.

Figure 2-39 Data forwarding statistics

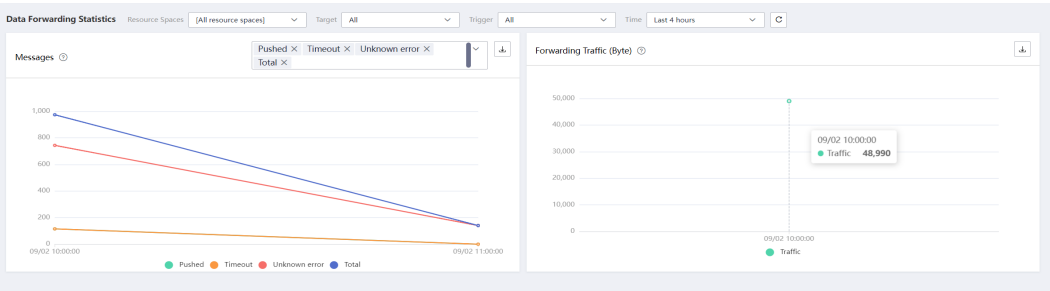


Table 2-10 Statistical report description

Report Name	Description	Statistical Period
Device Overview	The numbers of all devices, online devices, offline devices, frozen devices, and inactive devices, which indicate the numbers and statuses of devices in real time.	1 minute
Device Statuses	The numbers of online devices, offline devices, frozen devices, and inactive devices in the target resource space at a certain time of the day.	1 minute
Device Messages	The number of device messages of a specified resource space or all resource spaces of a tenant. The statistics are displayed by hour. The seven types of device messages are as follows: Message delivery, asynchronous command delivery, command delivery, event delivery, property reporting, event reporting, and message reporting	1 minute
API Calls	The total number of API calls and the number of 2xx, 4xx, and 5xx response codes in the specified resource space are collected every 5 minutes. The response codes are described as follows: Status code 2xx indicates that the server has successfully processed the request from the client. Status code 4xx indicates that the client request is incorrect. For example, the client requested a non-existent page or the client provided invalid authentication information. Status code 5xx indicates that the server failed to process the request due to an error. For example, request processing on the server timed out. You can query API call statistics in the last 4 hours, last day, last 3 days, or last week.	5 minutes
Messages	IoTDA collects the total number of messages of the corresponding target and trigger in the current resource space every hour. You can select a resource space, target, and trigger monitored to view detailed message statistics. The statistics can be filtered by status (push success, timeout, unknown error) and total number.	1 minute
Forwarding Traffic	IoTDA collects the traffic (in bytes) of successful data forwarding of the corresponding target and trigger in the current resource space every hour.	1 hour

2.4.2 Device Alarms

IoTDA provides the device alarm management functions, including reporting alarm by devices, reporting and clearing alarms by setting **device linkage** rules,

viewing device alarms, collecting statistics on device alarms, and manually clearing device alarms. You can filter the alarms to be viewed by setting the filter criteria, such as the device ID, product name, and alarm severity, and alarm status. Pay close attention to the device alarms and handle them in a timely manner to ensure the normal device running.

Configuring a device linkage rule to report alarms

You can set a **device linkage** rule, set the response action to alarm reporting, and define the alarm attributes and alarm severity on IoTDA. When the trigger conditions are met, IoTDA reports an alarm. For example, when the battery level of a smart water meter is lower than 10%, IoTDA generates an alarm for low battery level. Maintenance personnel can locate the water meter based on the alarm information and replace the battery.

Viewing alarm information

You can view alarm information in a specified period.

Step 1 Log in to the IoTDA console.

Step 2 In the left navigation pane, choose **O&M > Alarms**. On the page that is displayed, view the alarm information about IoTDA.

Step 3 Locate the target alarm in the list and click its name to check its details.

Step 4 Clear the alarm. After a device fault is rectified, you can click the clear button in the **Operation** column of the target alarm in the alarm list to manually clear the alarm.

----End

2.4.3 Message Trace

Scenario

If a fault occurs in core service scenarios (device authentication, device alarm reporting, device status change, command delivery, data reporting, or data forwarding), IoTDA can use the message trace function to quickly locate the fault and analyze the cause. Currently, message tracing is not applicable to other scenarios. IoTDA supports message trace for NB-IoT and MQTT devices. You can trace messages for up to 50 devices simultaneously. The maximum number of message trace records that can be displayed and the maximum number of records can be exported for a single device depend on the queue capacity. The maximum queue capacity is 500 or 2,000 records.

Procedure

Step 1 Log in to the IoTDA console.

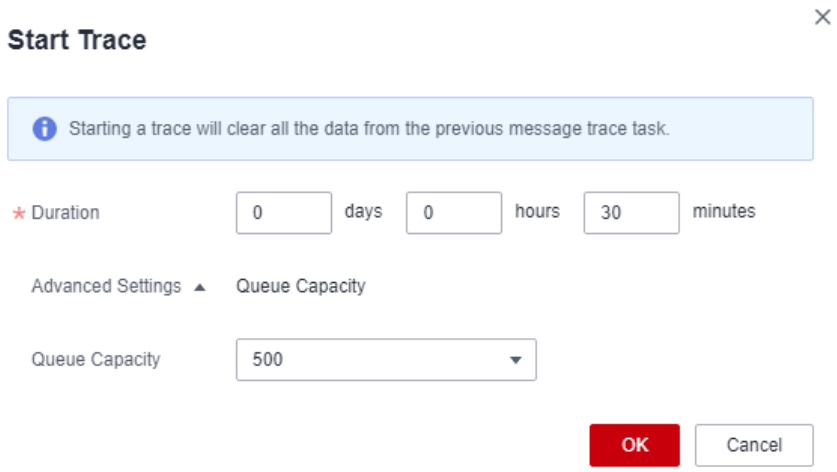
Step 2 In the left navigation pane, choose **Devices** > **All Devices**.

Step 3 Search for the device to trace and click **View** to access its details.

Step 4 On the **Device Details** page, click the **Message Trace** tab, and click **Start Trace** to set the message tracing duration. The message tracing duration indicates the duration from the time when message tracing starts to the time when message

tracing ends. Messages are traced in this duration. After the tracing configuration is modified, the new time is used. You can expand the **Advanced Settings** to set the trace queue capacity (the maximum number of records can be saved for this message trace task).

Figure 2-40 Starting message tracing



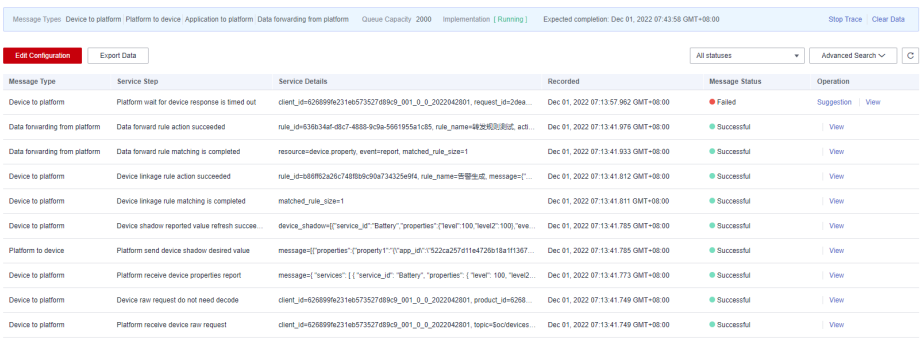
Step 5 View the services that are being traced on the **Message Trace** tab page of the **Device Details** page. You can also click **Stop Trace** to stop the task. If there are many records in the trace result, you can filter them by message status, message type, record time, keyword of service details, and flow ID. The relationship between multiple keywords of service details is logical AND. You can click **Export Data** to export the trace result for further analysis. If the message status is **Failed**, you can click **View** to view the result details and locate the fault based on the failure handling suggestions.

NOTE

To enable message tracing for child devices, you need to enable message tracing for the corresponding gateway. Otherwise, some message tracing data will be lost.

The maximum number of message trace records that can be displayed and the maximum number of records can be exported for a single device depend on the queue capacity. The maximum queue capacity is 500 or 2,000 records.

Figure 2-41 Viewing message trace results



----End

2.4.4 Online Debugging

Overview

After the product models and codecs are developed, the application can receive data reported by the device and deliver commands to the device through IoTDA.

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

Debugging a Product by Using a Virtual Device

When both device development and application development are not completed, you can create virtual devices and use the application simulator and device simulator to test product models and codecs.

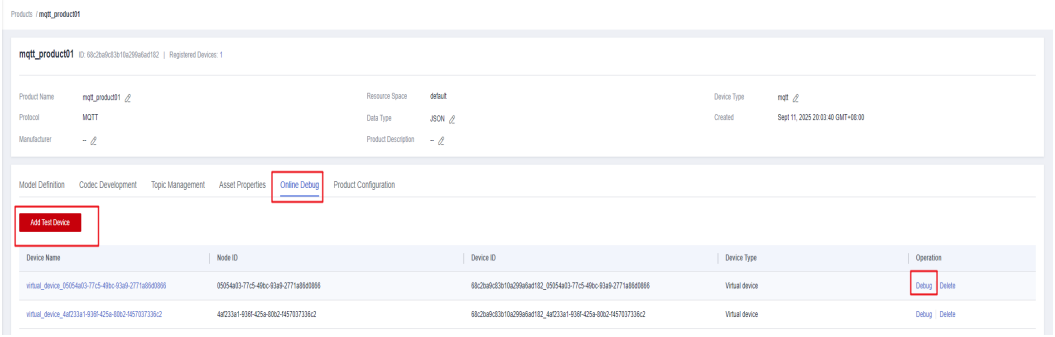
Step 1 Log in to the IoTDA console.

Step 2 Choose **Products** in the left navigation pane.

Step 3 Search for the product to debug and click **Debug** to access its details. On the product details page, click the **Online Debug** tab and perform the following operations based on whether a new virtual test device is used for debugging:

- If yes, go to **Step 4**.
- If no, go to **Step 5**.

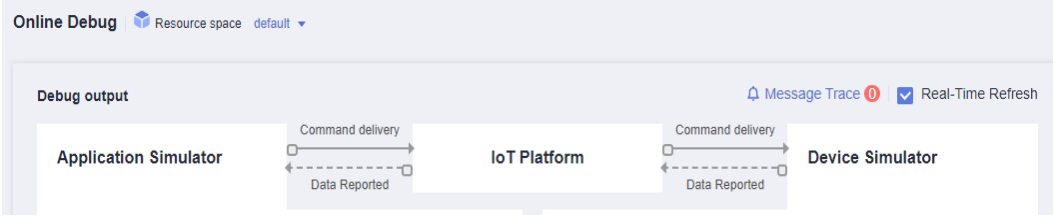
Figure 2-42 Adding or selecting a virtual test device



Step 4 Click **Add Test Device**. In the displayed dialog box, select **Virtual device** and click **OK** to create a virtual device.

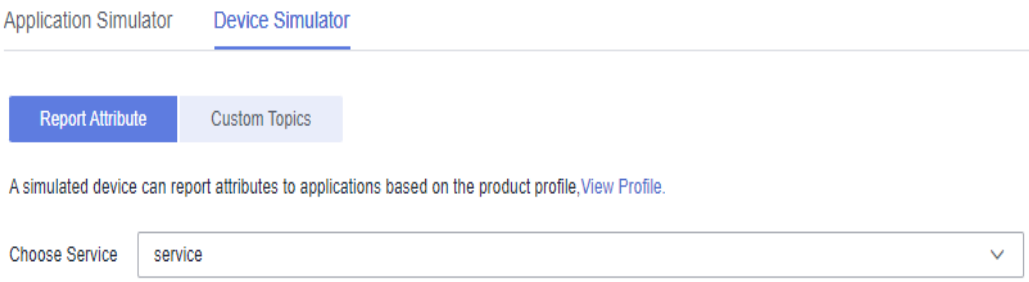
Step 5 In the device list, select the virtual device to debug and click **Debug** to go to the **Online Debug** page.

Figure 2-43 Online debugging information



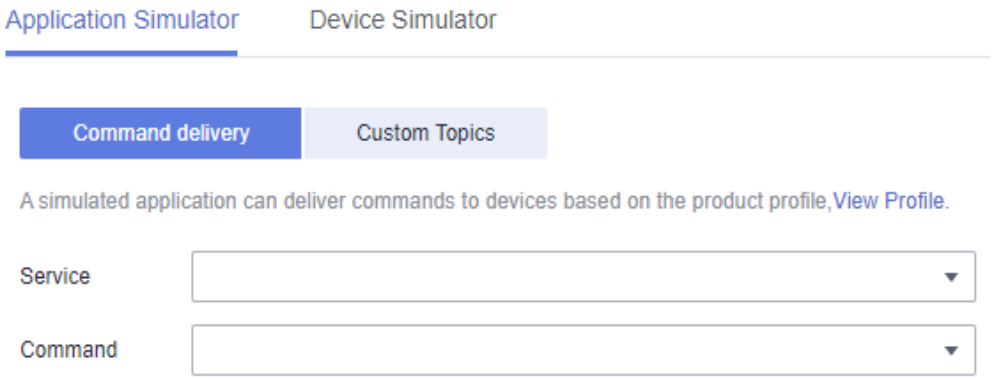
Step 6 For an MQTT device, under the **Device Simulator** tab, select **Report Attribute** and enter the corresponding property values, or select **Custom Topics**, enter a topic name and message content, and click **Send**. View the data reporting result in the online debugging information area and IoTDA processing logs in **Message Trace**.

Figure 2-44 MQTT device simulator



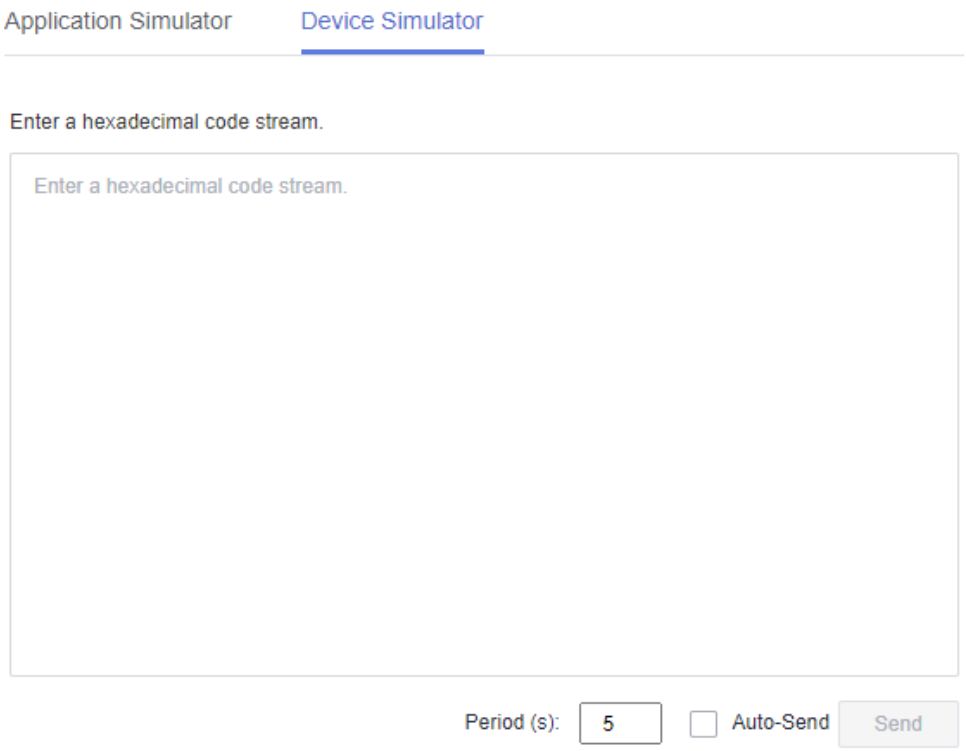
Step 7 For an MQTT device, under the **Application Simulator** tab, select **Command delivery** and enter the corresponding command, or select **Custom Topics**, enter a topic name and message content, and click **Send**. View the command delivery result in the online debugging information area and IoTDA processing logs in **Message Trace**.

Figure 2-45 MQTT application simulator



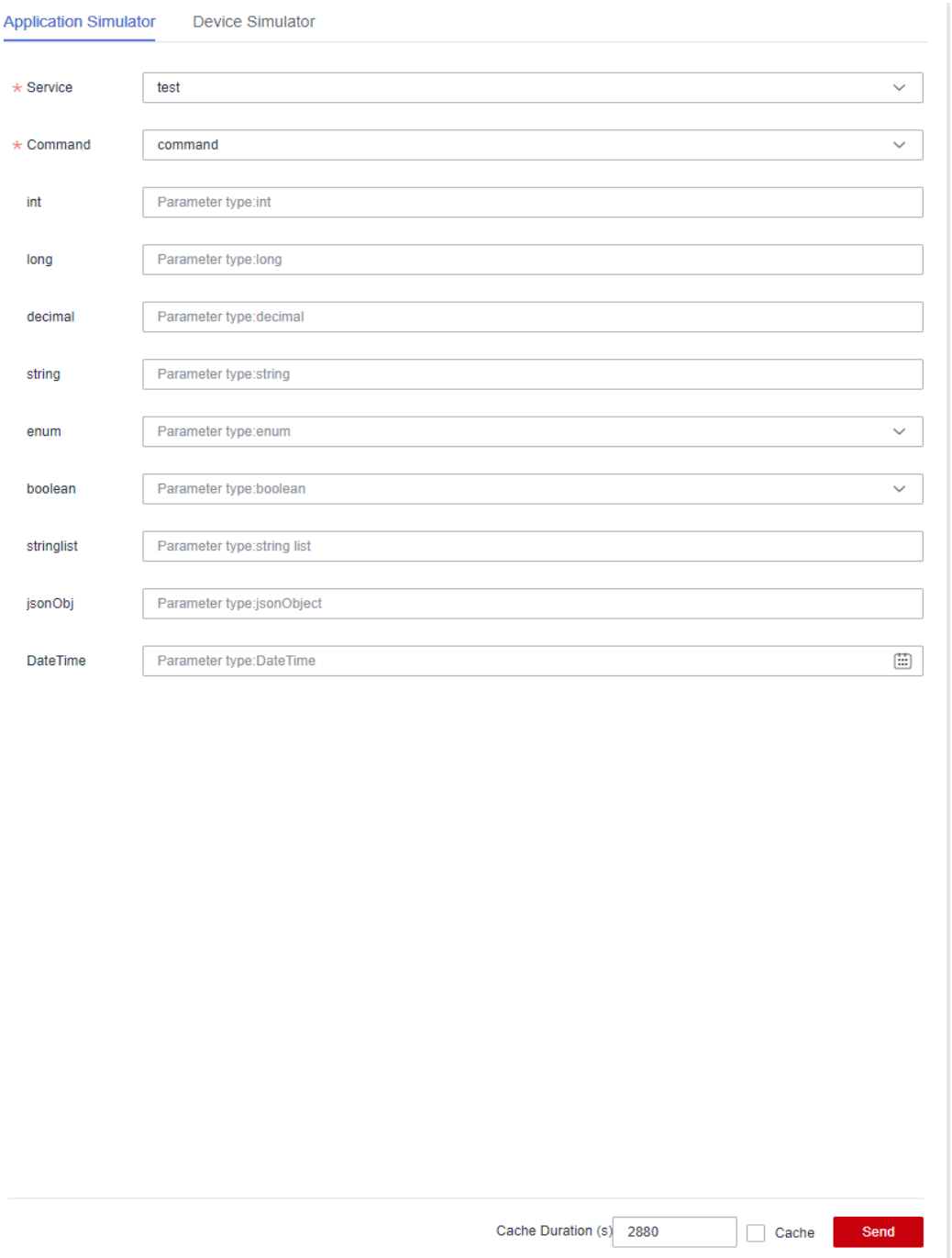
Step 8 For a CoAP device, under the **Device Simulator** tab, enter a hexadecimal code stream of the data and click **Send**. View the data reporting result in the online debugging information area and IoTDA processing logs in **Message Trace**.

Figure 2-46 CoAP device simulator



Step 9 For a CoAP device, under the **Application Simulator** tab, enter a command value, select the option of cached sending and then click **Send**, or directly click **Send**. View the data reporting result in the online debugging information area and IoTDA processing logs in **Message Trace**.

Figure 2-47 CoAP application simulator



----End

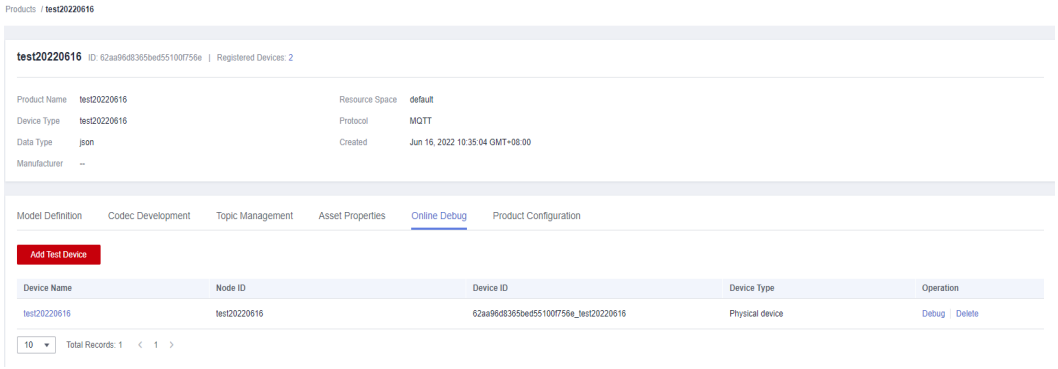
Debugging a Product by Using a Physical Device

When the device development is complete but the application development is not, you can add physical devices and use the application simulator to test devices, product models, and codecs.

Step 1 On the product details page, click the **Online Debug** tab, and perform the following operations based on whether a registered physical device is used for debugging:

- If no, go to **Step 4**.
- If yes, go to **Step 5**.

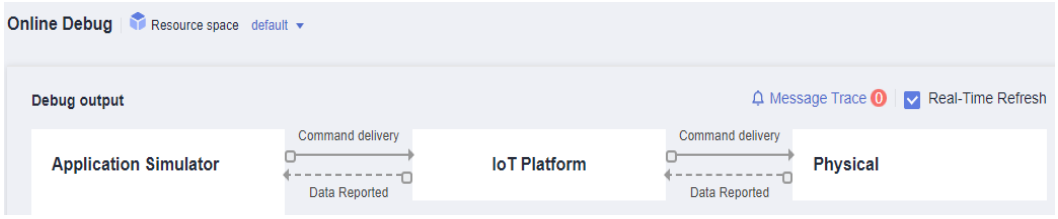
Figure 2-48 Adding or selecting a physical test device



Step 2 Click **Add Test Device**. In the displayed dialog box, select **Physical device**, enter the device information, and click **OK**.

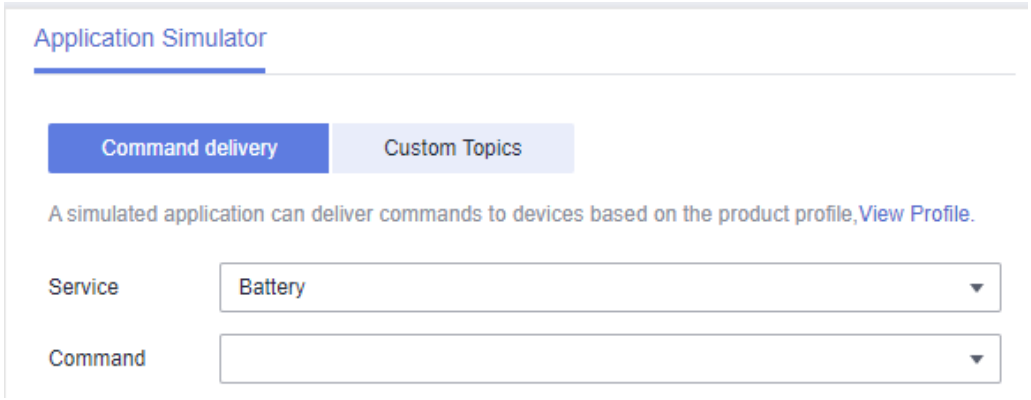
Step 3 In the device list, select the physical device to debug and click **Debug** to go to the **Online Debug** page.

Figure 2-49 Debugging interface



Step 4 For an MQTT device, under the **Application Simulator** tab, select **Command delivery** and enter the corresponding command, or select **Custom Topics**, enter a topic name and message content, and click **Send**. View the command delivery result in the online debugging information area and IoTDA processing logs in **Message Trace**.

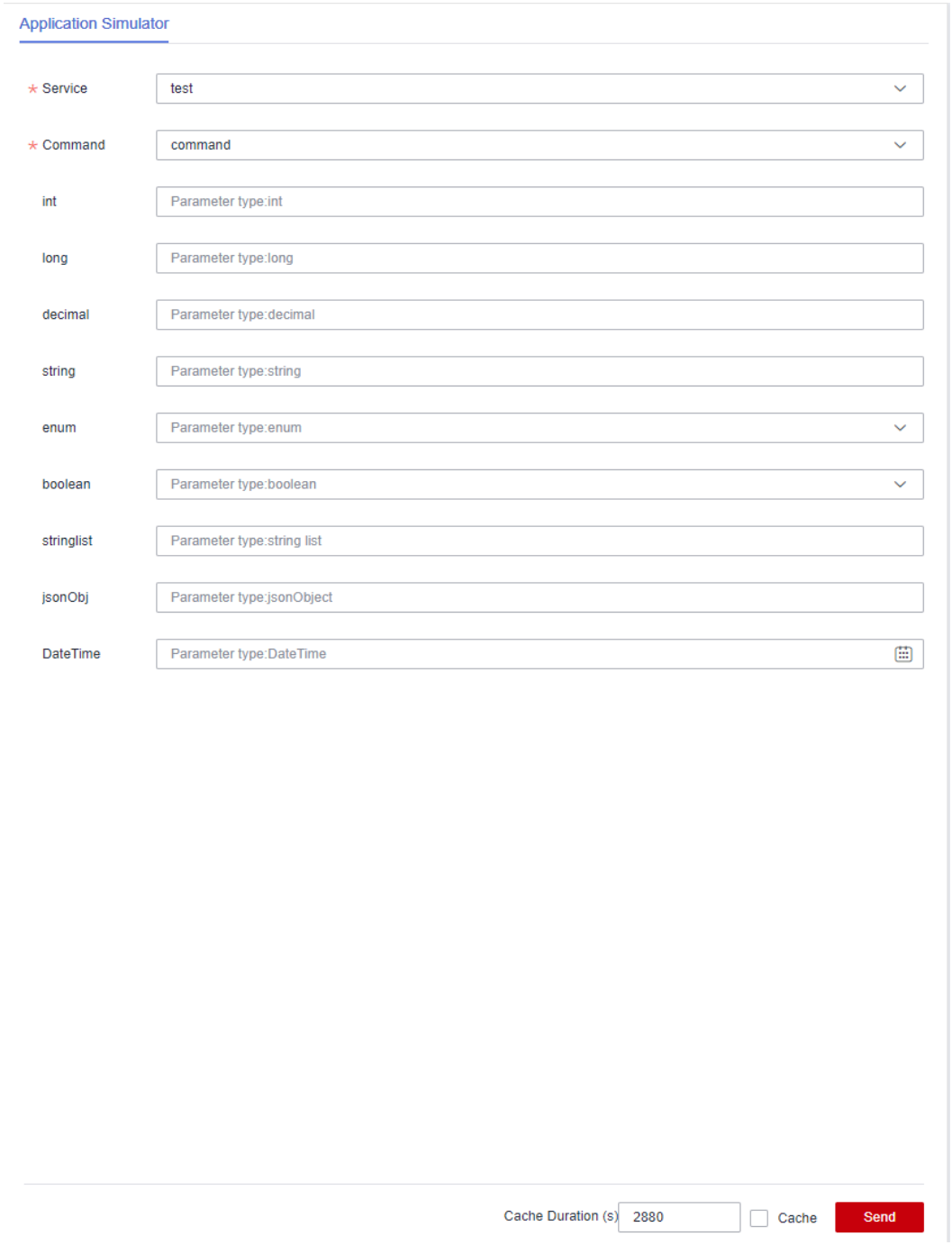
Figure 2-50 MQTT application simulator



Step 5 For a CoAP device, under the **Application Simulator** tab, enter a command value, select the option of cached send and then click **Send**, or directly click **Send**. View

the data reporting result in the online debugging information area and IoTDA processing logs in **Message Trace**.

Figure 2-51 CoAP application simulator



----End

2.5 Device Security Center

IoTDA provides the device security center, which provides security detection and offline analysis.

Security Detection

IoTDA continuously detects device security threats. This section describes security check items and how to view and handle detected security risks.

NOTE

The following detection items can be used only after being enabled: malicious IP addresses, memory leakage, abnormal ports, brute-force cracking login, file tampering, Common Vulnerabilities and Exposures (CVEs), malicious files, abnormal processes, insecure functions, and insecure protocols. Abnormal port/Malicious IP address checks: Enter whitelisted ports or IP addresses for checks. The system compares the parameters reported by the device and the configured whitelist members. You can add IP address segments to the whitelist, for example, 192.168.1.10/24. You need to configure a blacklist for insecure protocol detection. The platform will compare the protocols reported by the device against this blacklist. If any of the protocols used by the device are found in the blacklist, an anomaly event will be triggered.

Table 2-11 Check item description

Item	Description
Connection mode	No encryption protocol is used to establish secure connections between devices and IoTDA. This may cause man-in-the-middle and replay attacks and affect services.
Multiple connection establishments within a unit time	If a device attempts to establish connections with IoTDA for multiple times within 1 second, the device may be cracked with brute force. As a result, identity information may be leaked, normal devices may be forced to go offline, and service data may be stolen.
Cryptographic algorithm suite	Currently, IoTDA checks the following insecure cryptographic algorithm suites: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA, TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA, TLS_PSK_WITH_AES_128_CBC_SHA, TLS_PSK_WITH_AES_256_CBC_SHA Insecure cryptographic algorithm suites have security vulnerabilities, which may cause security risks such as device data leakage.
Device authentication failure	Incorrect device identity authentication information causes device connection failures. This may affect services.
Malicious IP address	Whether the device communicates with malicious IP addresses.
Memory leak check	Whether memory leaks occur on the device.
Abnormal port	Whether abnormal ports are enabled on the device.

Item	Description
Brute-force cracking login	Whether attackers attempt to log in to the device through brute force cracking.
Device file tampering	Whether files in a specified directory of a device are tampered with.
Insecure secret	Whether the device secret is secure (combination of letters and digits). During device registration, if the secret contains only uppercase letters, lowercase letters, or digits, this event is triggered.
Insecure certificate	Whether the certificate validity period is less than the configured threshold (90 days by default).
Abnormal message	Whether the size of messages reported by the device exceeds the configured threshold (200 KB by default).
CVEs	Whether the CVEs exist on the device.
Malicious file	Whether malicious files exist on the device.
Abnormal process	Whether abnormal processes exist on the device.
Insecure function	Whether insecure functions are enabled on the device.
Insecure protocol	Whether the specified insecure protocols are used in the device.

Offline Analysis

IoTDA help you analyze device offline causes by collecting statistics on the offline time range and characteristics of offline devices.

Table 2-12 Offline analysis description

Cause	Description
Active offline by the device	The device sends an MQTT disconnect packet to IoTDA for disconnection.
Heartbeat timeout	The device does not comply with the MQTT protocol. It sends MQTT heartbeat packets to IoTDA within 1.5 times the configured heartbeat interval. As a result, IoTDA considers that the device connection is invalid and cuts off the connection according to protocol requirements. (Note: The heartbeat interval is specified when the device establishes a connection with IoTDA.)

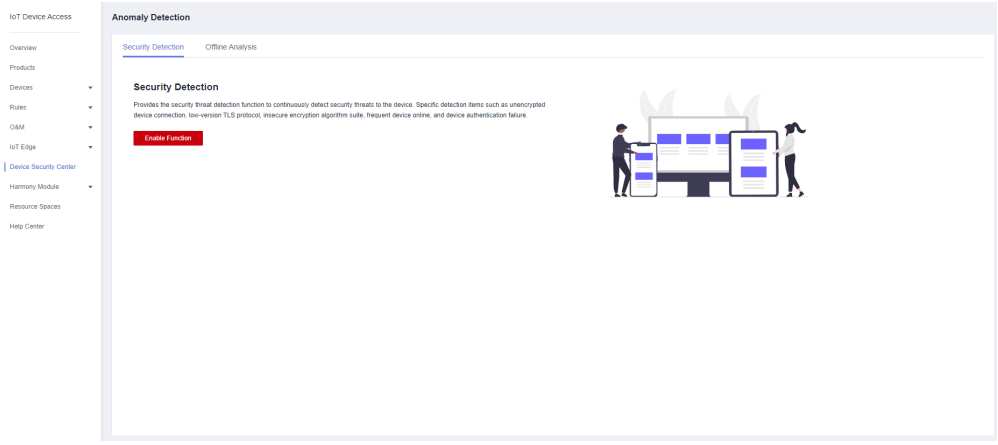
Cause	Description
Device-platform TCP connection cutoff	IoTDA receives a TCP disconnection packet from the device. As a result, the TCP connection between the device and IoTDA is cut off.
Device deleted	A tenant deletes a device from IoTDA, and IoTDA cuts off the connection with the device.
Device frozen	A tenant freezes a device on IoTDA, and IoTDA cuts off the connection with the device.
Connection cut off by the platform	IoTDA cuts off the connection with the device during upgrade.
Earlier connection cutoff	The device establishes connections with IoTDA repeatedly. IoTDA cuts off the existing connection and retains the new connection.
Device secret reset	When the device secret is reset and the connection is manually cut off, IoTDA cuts off the connection with the device.

Procedure

Step 1 Log in to the IoTDA console.

Step 2 In the navigation pane, choose **Device Security Center**. The **Anomaly Detection** page is displayed.

Figure 2-52 Anomaly detection overview



Step 3 **Security Detection** and **Offline Analysis** pages are available. Click **Enable Function** to enable the two functions. Otherwise, they cannot be used.

Figure 2-53 Security detection

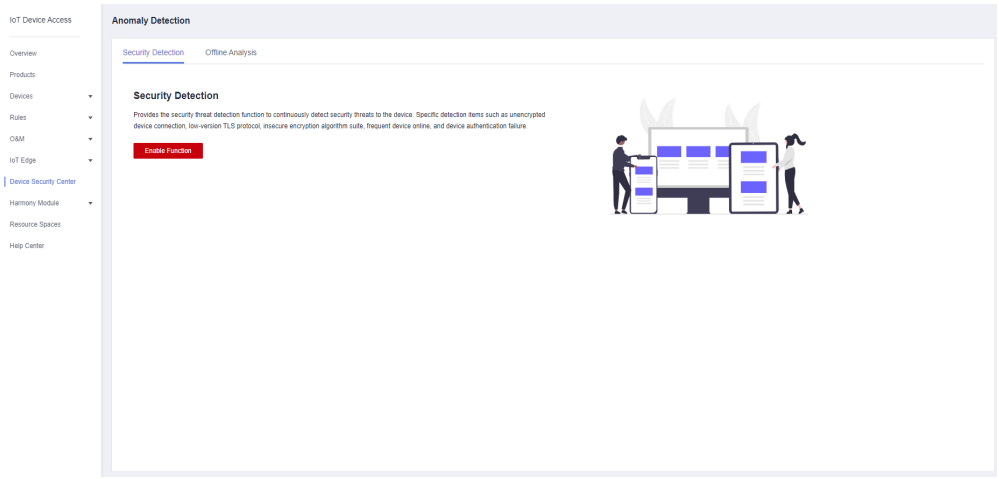
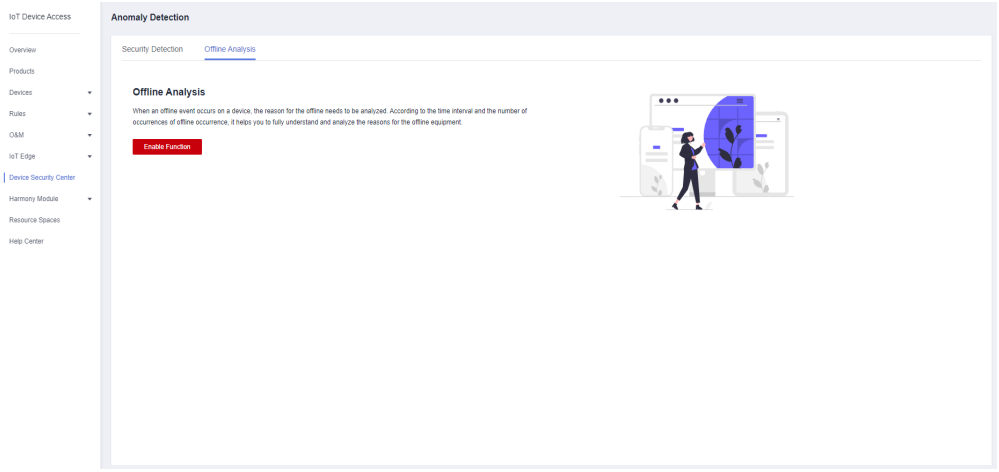


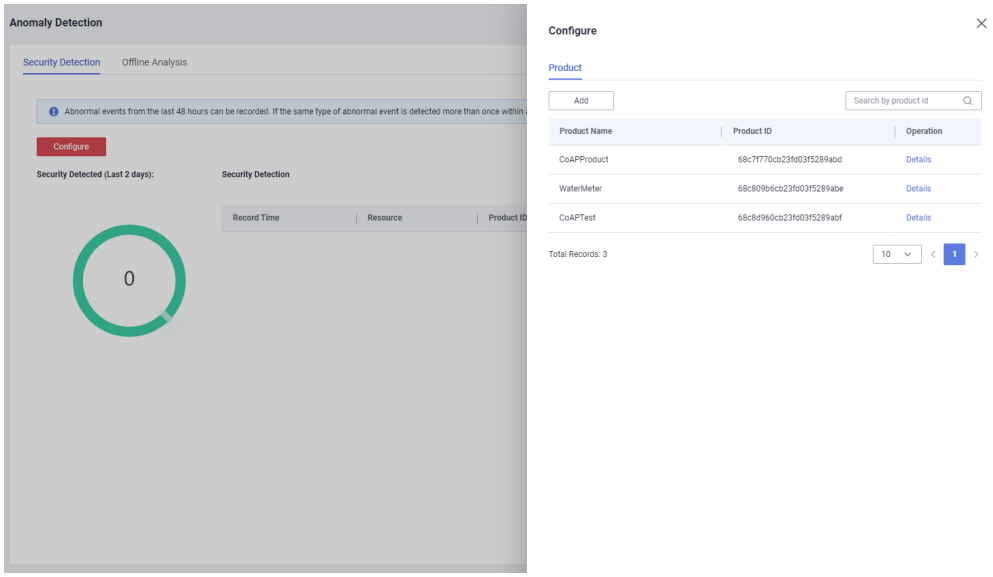
Figure 2-54 Offline analysis



Step 4 To enable **Security Checks**, perform the following steps. Otherwise, skip them.

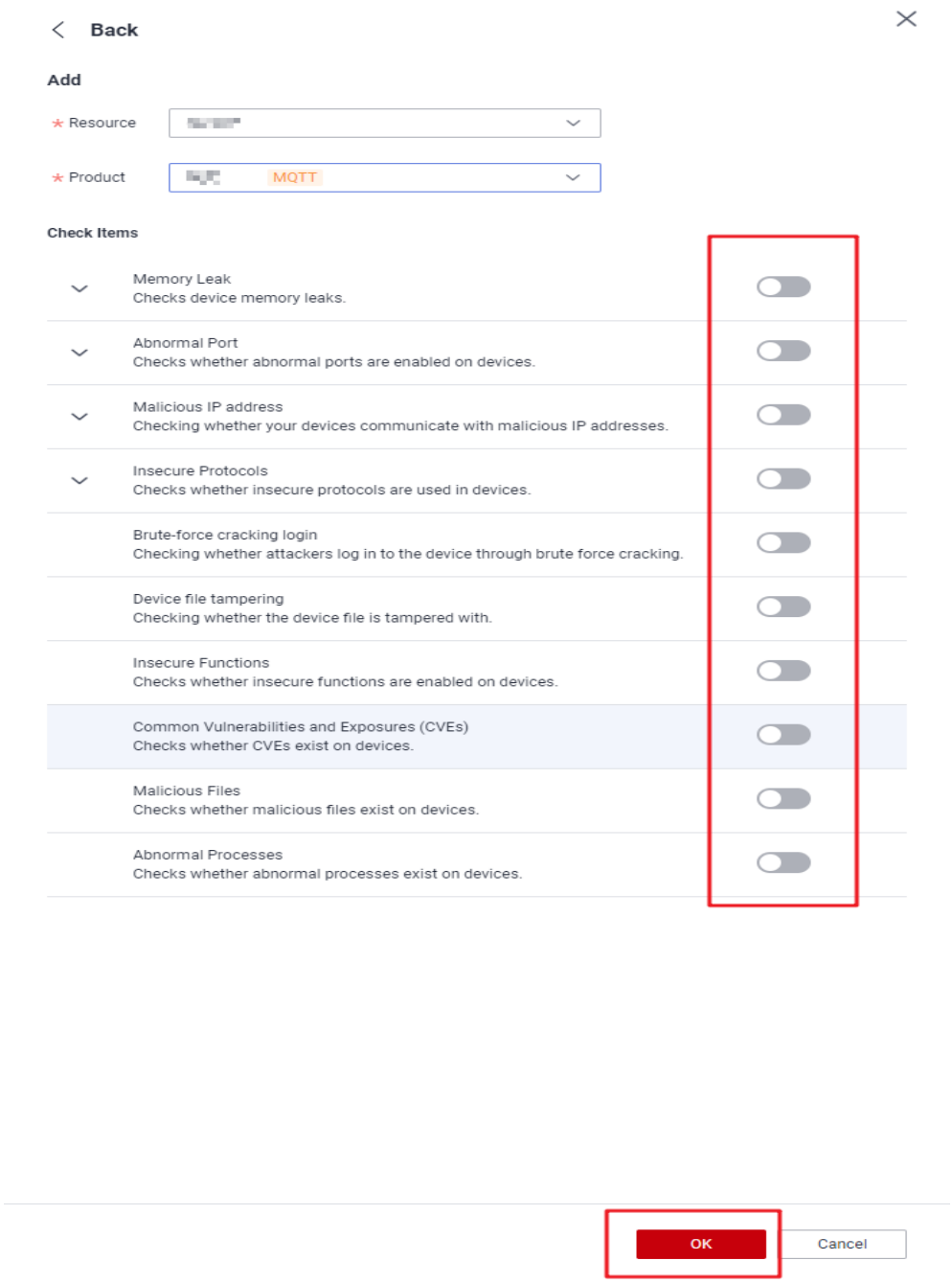
1. On the **Security Checks** tab page, click **Security Check Configuration**. In the displayed dialog box, click **Add**.

Figure 2-55 Anomaly detection - Security check configuration



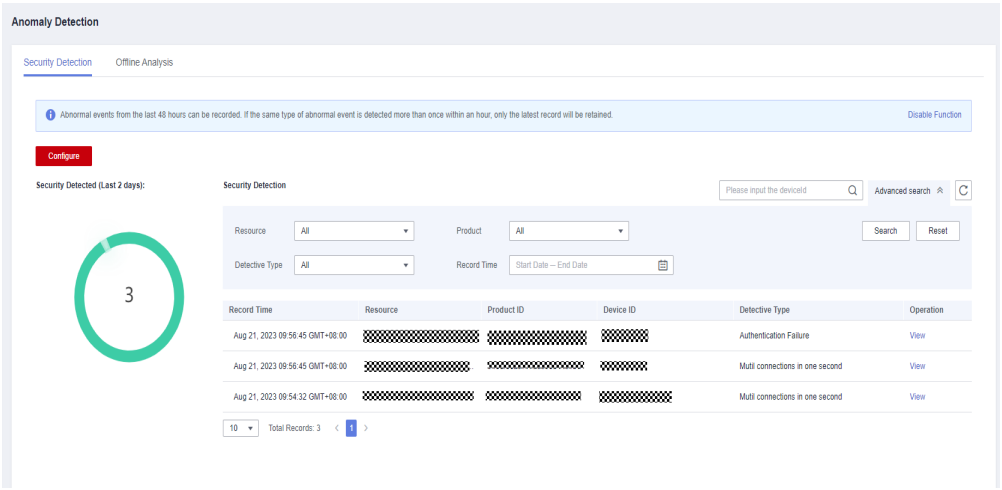
2. On the configuration page, select the resource space to which the product belongs and the name of the product, and enable items as required.

Figure 2-56 Anomaly detection - Security check item configuration



Step 5 After the configuration, IoTDA starts security detection on devices. Abnormal events of the last 48 hours can be stored at most. You can search for security detection records by device ID, resource space, product, check item, and time range.

Figure 2-57 Security detection overview

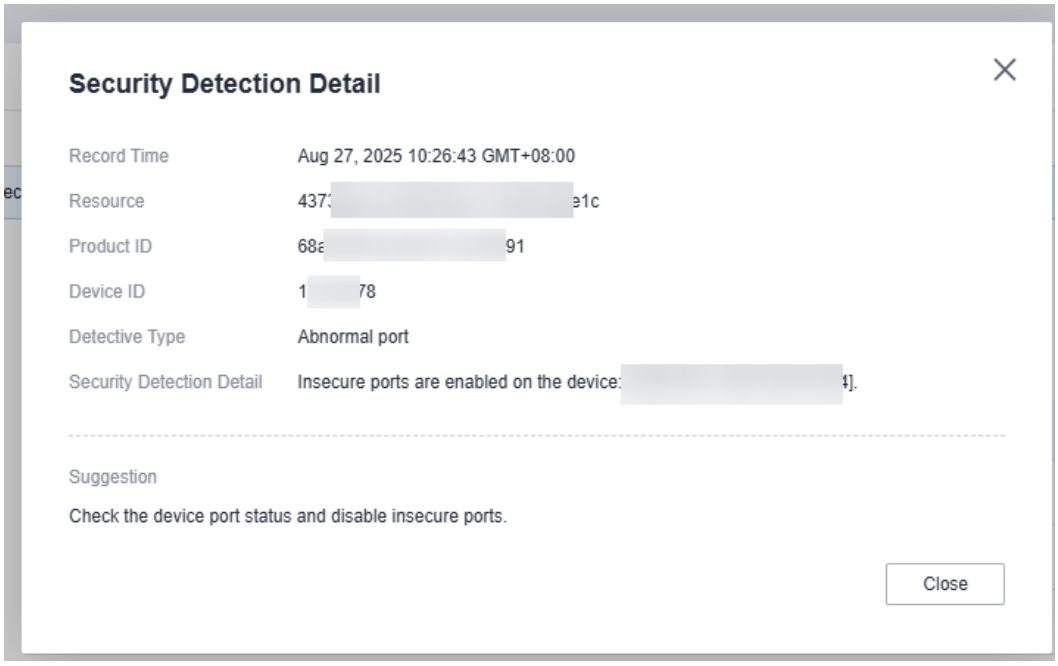


NOTE

1. Abnormal events in the last 48 hours can be recorded. If the same type of abnormal events are detected within 1 hour, only the latest record is retained.
2. Currently, offline analysis chart only supports aggregation of abnormal events by hour.
3. The abnormal event data follows a non-immediate write policy, causing a maximum 10-second delay between the device reporting the event and its stable display on the GUI. After the device reports an abnormal event for 10 seconds, it can be consistently queried and shown on the page.

Step 6 Click the button in the operation column on the left to check the detection item details.

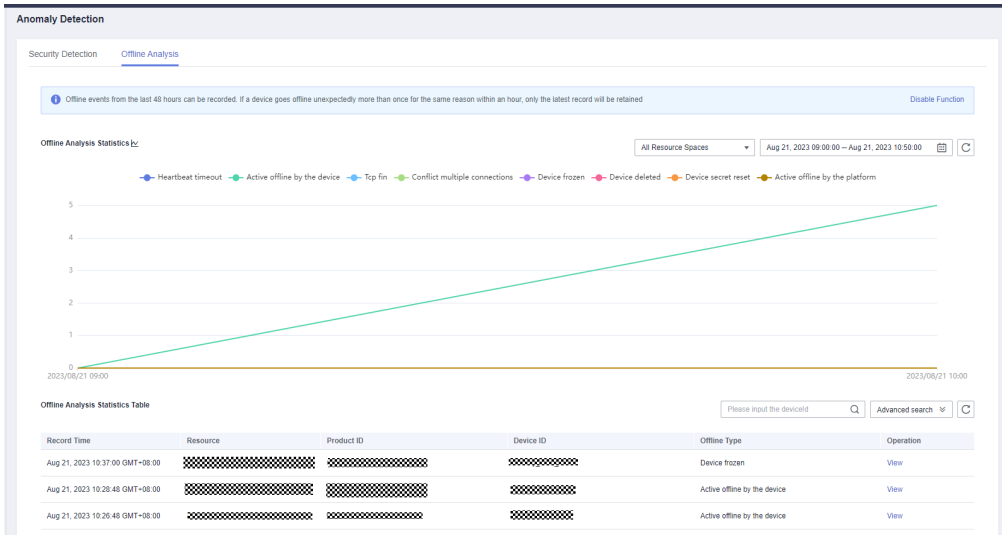
Figure 2-58 Security detection details



Step 7 After offline analysis is enabled, IoTDA automatically analyzes the causes of device disconnections. Disconnection events of the last 48 hours can be stored at most.

You can search for the records by device ID, resource space, product, cause, and time range. You can set the chart to display data by resource space and time range.

Figure 2-59 Offline analysis overview

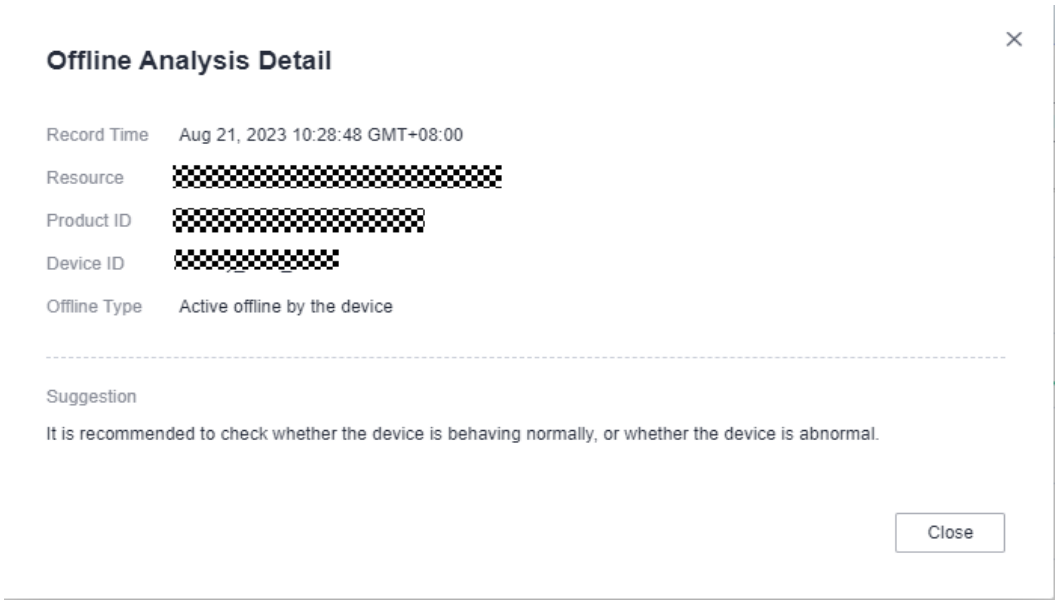


NOTE

Abnormal events in the last 48 hours can be recorded. If the same type of offline events are detected within 1 hour, only the latest record is retained.

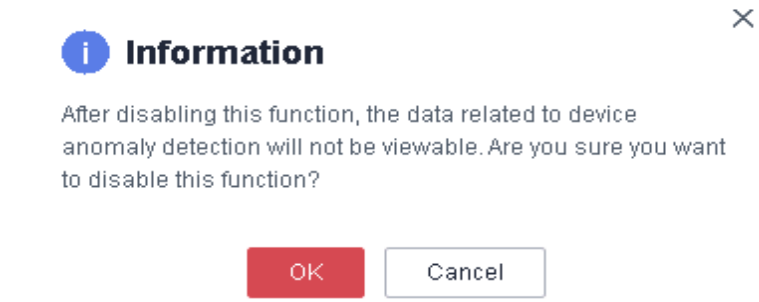
Step 8 Click **View** to view details about each disconnected device in the list.

Figure 2-60 Offline analysis details



Step 9 Click **Disable** to disable security detection and offline analysis. Then, these functions are unavailable. To use these functions, enable them again.

Figure 2-61 Prompt for disabling the function



----End

2.6 HarmonyOS Module

2.6.1 HarmonyOS Soft Bus

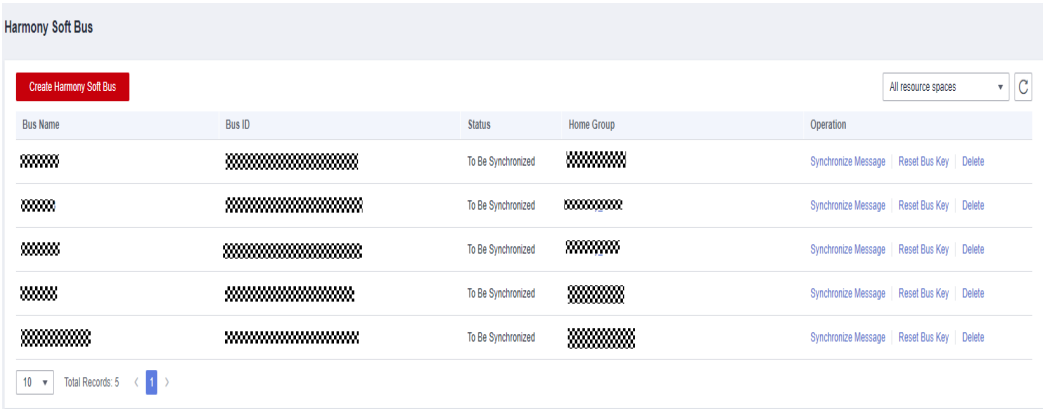
This page displays a list of all Harmony soft buses and the data can be filtered by resource space. You can create and delete Harmony soft buses, synchronize messages, and reset keys for Harmony soft buses, facilitating the unified management.

Procedure

Step 1 Log in to the IoTDA console.

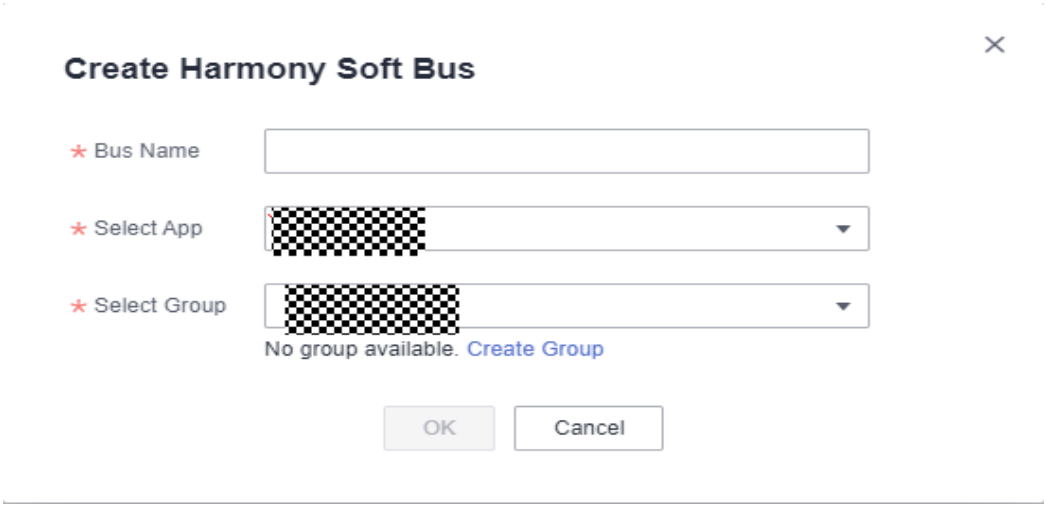
Step 2 In the navigation pane, choose **Harmony Module** > **Harmony Soft Bus** to query the list.

Figure 2-62 Overview of Harmony soft buses



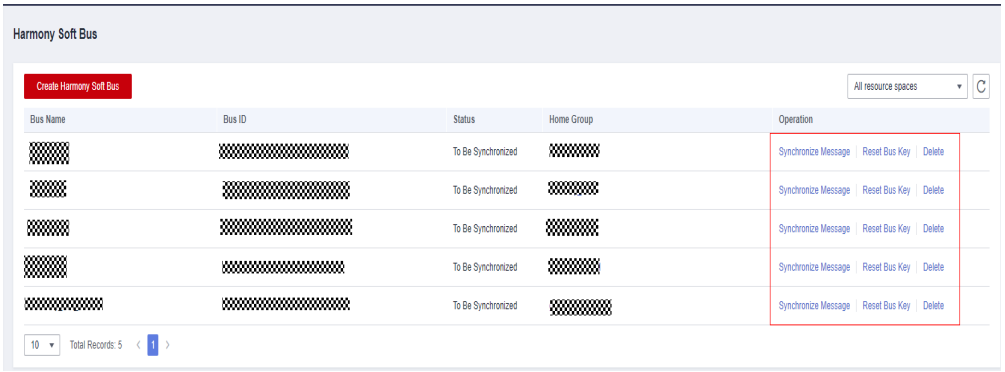
Step 3 Click **Create Harmony Soft Bus**.

Figure 2-63 Creating a Harmony soft bus



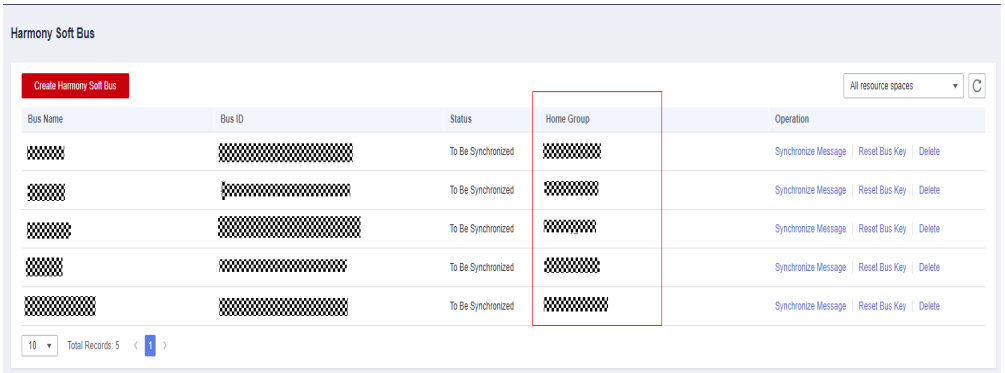
Step 4 You can perform operations on the Harmony soft bus in the **Operation** column.

Figure 2-64 Operation column



Step 5 Click a group name in the **Home Group** column to go to the corresponding group details page and perform operations. For details, see **Procedure**.

Figure 2-65 Group operations



----End

2.6.2 Device Engine

This page is dedicated for device engines. The prerequisites are as follows:

- Device-side rules support only command delivery actions.
- Devices must be integrated with IoT Device SDK (C) v1.1.2 or later.
- Devices need to report the SDK version number to IoTDA using the APIs provided by the SDK.

For details, see **Device-side Rules**.

2.7 Resource Spaces

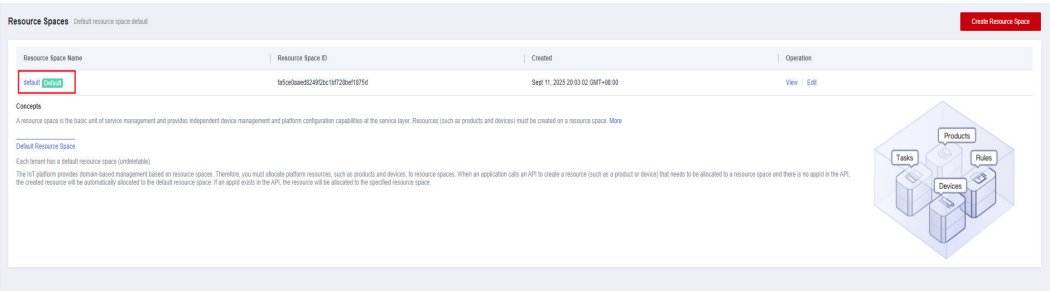
The resource space provides independent device management and platform configuration capabilities at the service layer. As the basic unit of service management, resources (such as products and devices) created on the platform must belong to a resource space. You can use the resource space for domain-based management of multiple service applications.

- You can create a maximum of 10 resource spaces on IoTDA. The space automatically created by the platform when the IoTDA service is subscribed to for the first time is the default resource space.
- An appId (parameter **app_id** in API calls), which is a unique identifier of a resource space, is allocated by the platform when the resource space is created.
- After a resource space is created, you can view its appId in the resource space.
- The default resource space cannot be deleted. After a resource space is deleted, all resources in the space, such as devices, products, and subscription data, are deleted from the platform and cannot be restored. Exercise caution when deleting a resource space.

Creating a Resource Space

When you subscribe to IoTDA for the first time, IoTDA automatically creates the default resource space. You have only one default resource space, which cannot be deleted.

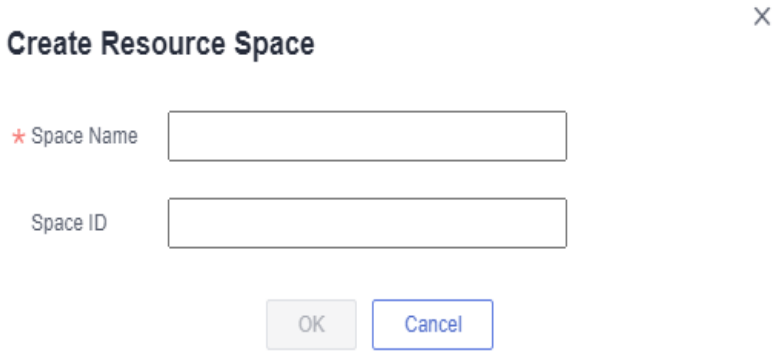
You can create a product or register a device in the default resource space. You can also perform the following steps to create a resource space:



Step 1 Log in to the IoTDA console.

Step 2 In the left navigation pane, choose **Resource Spaces**. In the upper right corner, click **Create Resource Space**. On the page displayed, set **Space Name** and click **OK**.

The resource space name must be unique under the account. It can be changed after the resource space is created.



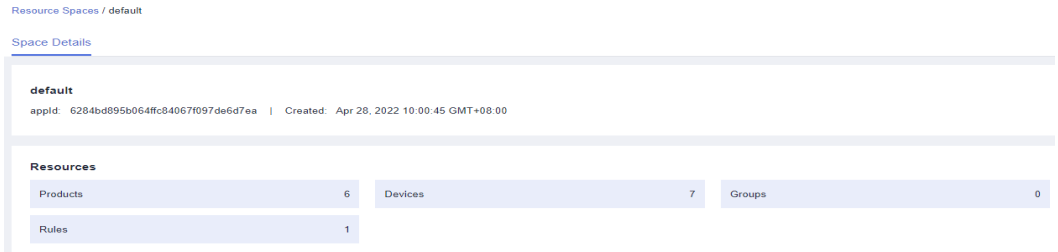
----End

NOTE

When creating a resource space, you can specify a resource space ID and it must be globally unique. If a resource space ID already exists, the system will display a message that indicates the ID is duplicate. If you deleted a resource space, do not create another resource space with the same ID as the deleted one within 48 hours. This is because it takes up to 48 hours for associated data in the deleted resource space to be completely cleared. During this period, resource spaces with the same ID cannot be created.

Querying a Resource Space

After a resource space is created, you can click **View** in the **Operation** column to view the APPID, creation time, and numbers of products, devices, groups and created rules of the resource space.



2.8 Plug-ins

2.8.1 Introduction

In scenarios such as smart city, campus, and manufacturing, some industry devices need to be directly connected to the platform. The communication between these devices and the platform complies with industry protocol standards, such as JT808, HJ212, and SL651. The platform supports access of these TCP-based devices through the plug-in extension mechanism.

After the plug-in developed based on the platform specifications is uploaded and deployed on the console, industry protocol devices can be directly connected to the platform.

2.8.2 Procedure

Step 1 Log in to the IoTDA console.

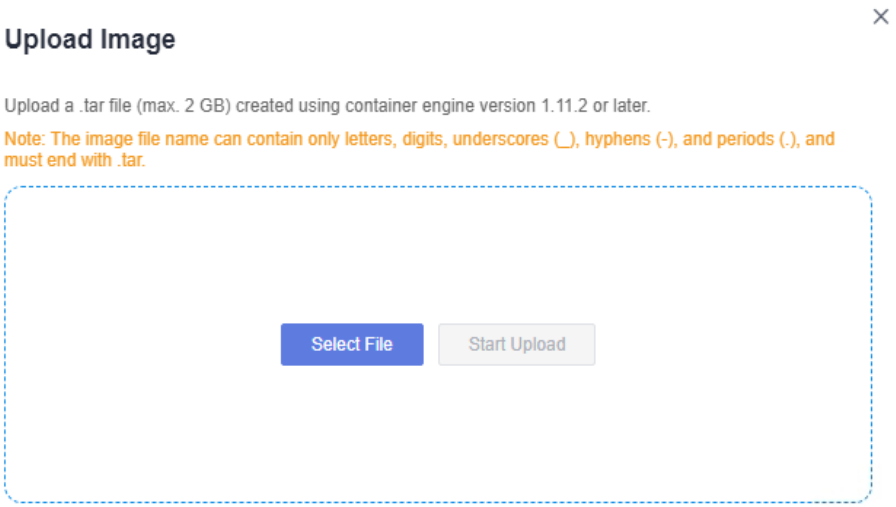
Step 2 Choose **Plug-in Management** in the left navigation pane.

Step 3 Check whether the plug-in type is hosted or non-hosted. For hosted plug-ins, create and upload an image in advance by referring to the section about generic-protocol development in *Developer Guide*. For non-hosed plug-ins, create a plug-in, obtain the bridge ID and bridge secret on the page, and perform offline development. Subsequent steps including image upload are not required.

Step 4 For hosted plug-ins, click the button for uploading an image to upload the plug-in image in advance. For details about how to create an image, see related guide in the section of generic-protocol development in *Developer Guide*.

Step 5 Image upload: Click **Upload Image** to upload the package of the plug-in image to deploy. The image file name can contain only letters, digits, underscores (_), hyphens (-), and periods (.), and ends with .tar. After the image is uploaded, it will be pushed to the image repository. You can click **Image List** next to **Upload Image**. View the image push status. (By default, each tenant can upload up to 10 image files. To increase the quota, contact the administrator.)

Figure 2-66 Previous image

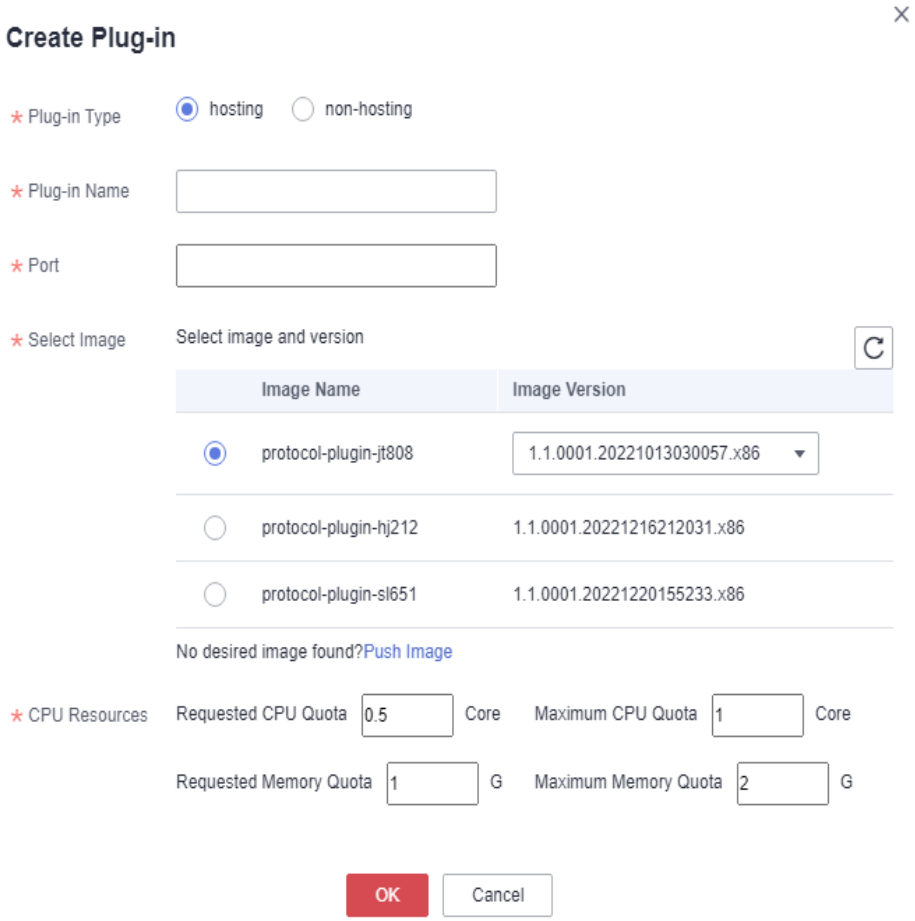


Step 6 Operations on plug-ins:

- Creation: Click **Create Plug-in**. In the displayed dialog box, enter the information and click **OK**.
  a. Hosted plug-ins (deployed by the platform)

     To create a hosted plug-in, specify the plug-in name, port number, image, and CPU resources for the platform to deploy the plug-in.

Figure 2-67 Creating a plug-in



b. Non-hosted plug-ins (deployed by users)

Non-hosting plug-in deployment means that you need to apply for servers to deploy generic-protocol plug-ins instead of deploying generic-protocol plug-ins on the IoT platform. The plug-ins you deployed are used to connect generic-protocol devices to the IoT platform through cloud-to-cloud interworking.

Choose **Plug-in Management**, click **Create Plug-in**, set **Plug-in Type** to **non-hosting**, enter a plug-in name, and click **OK**. Obtain the bridge ID and bridge secret.

Figure 2-68 Creating an unhosted plug-in

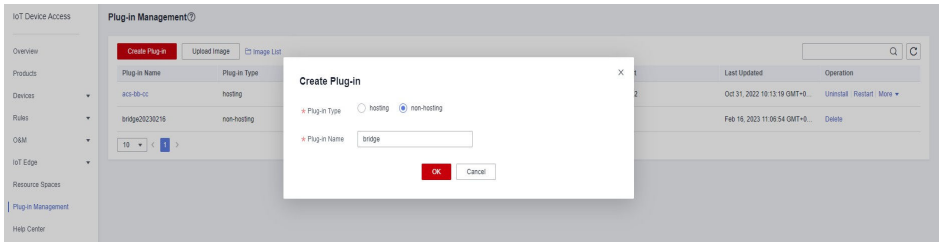
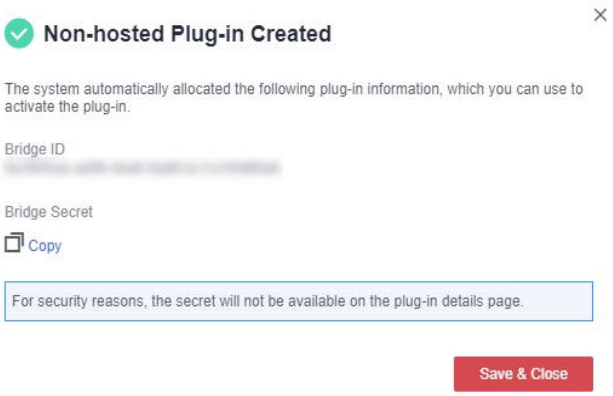
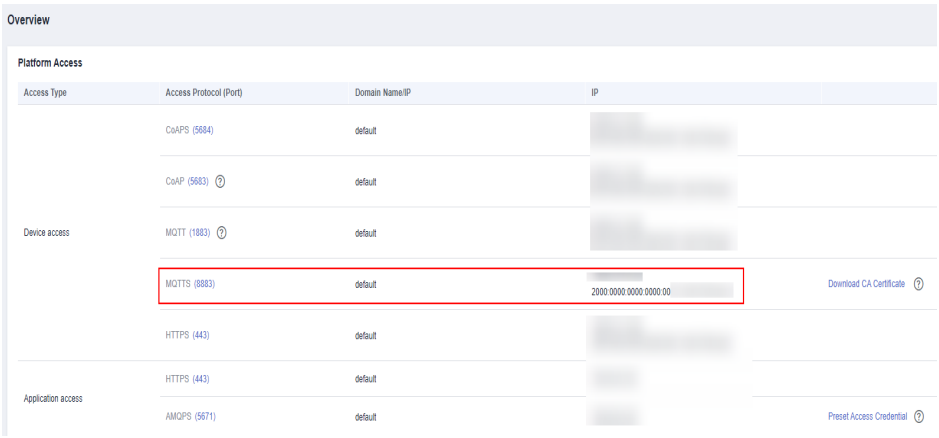


Figure 2-69 Non-hosted plug-in created



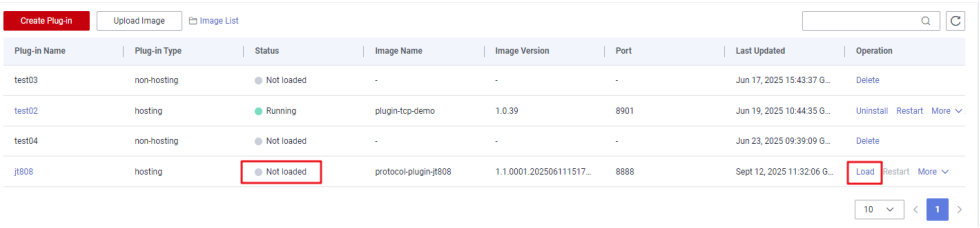
After obtaining the bridge ID and bridge secret, inject them into the generic-protocol plug-in as environment variables. On the **Overview** page, use the address corresponding to the MQTTS access protocol to connect devices to the IoT platform. Then, the generic-protocol plug-in can be deployed in non-hosting mode. Currently, the generic-protocol plug-in can only establish connections with the IoT platform using the address corresponding to the MQTTS access protocol.

Figure 2-70 Checking the address corresponding to the MQTTS access protocol



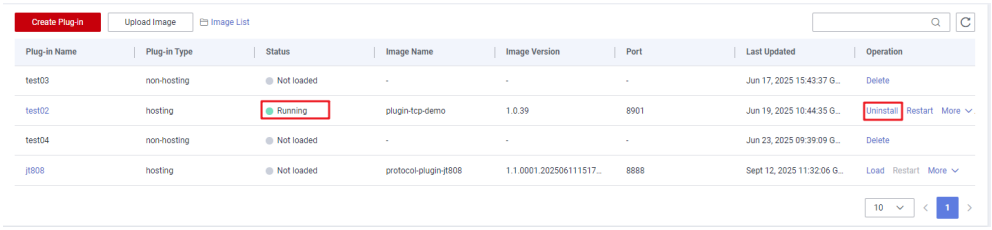
- Loading: Deploy a plug-in. On the right of the plug-in list, click **Load** to deploy the plug-in. (Only hosted plug-ins that are not loaded can be loaded.)

Figure 2-71 Loading a plug-in



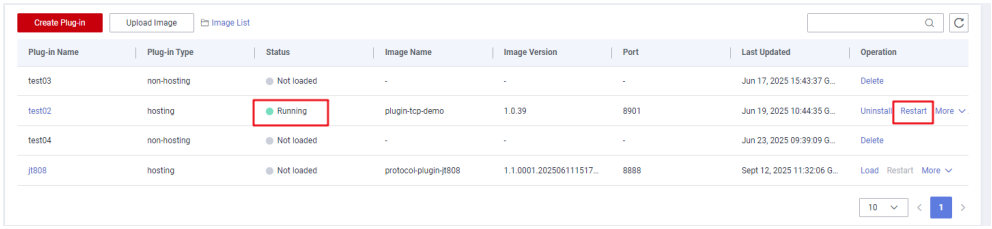
- Uninstallation: Stop a deployed plug-in. On the right of the plug-in list, click **Uninstall** to stop the plug-in. (Only hosted plug-ins in the running or abnormal state can be uninstalled.)

Figure 2-72 Uninstalling a plug-in



- Restart: Restart a plug-in. On the right of the plug-in list, click **Restart** to restart the plug-in. (Only hosted plug-ins in the running or abnormal state can be restarted.)

Figure 2-73 Restarting a plug-in

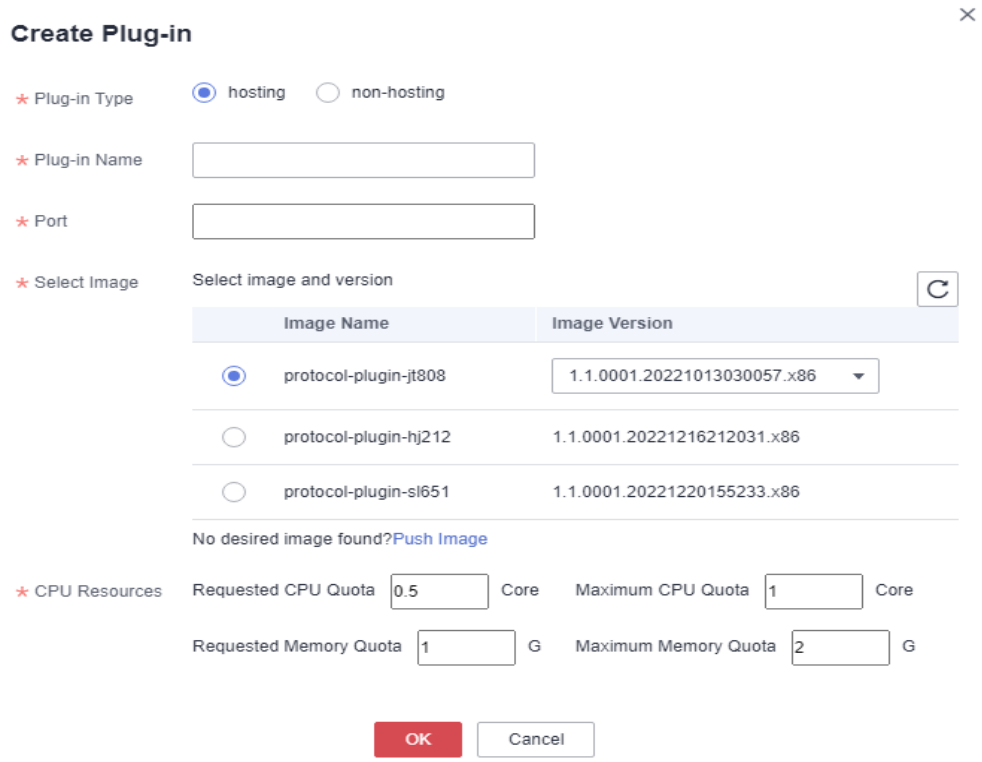


- Upgrade: Change the image version (only versions of the same image can be changed) and restart the plug-in. On the right of the list, choose **More** > **Upgrade**. (Only hosted plug-ins in the running or abnormal state can be upgraded.) Select an image version to upgrade the plug-in.

Figure 2-74 Upgrading a plug-in

- Delete: Delete all information about the plug-in. Click **Delete** to delete the plug-in. (You can only delete unloaded plug-ins.)
- Editing: Modify the plug-in deployment information, including the name, port, image, and CPU resources. The modified information takes effect only after the plug-in is restarted or deployed next time.

Figure 2-75 Modifying a plug-in



Step 7 Check deployment events (only for hosted plug-ins).

Check deployment events to locate faults when plug-in deployment or services are abnormal.

In the plug-in list, click the plug-in name. The plug-in details page is displayed. In the pod information area, the **View Deployment** button is available, as shown in the following figures.

Figure 2-76 Plug-in management

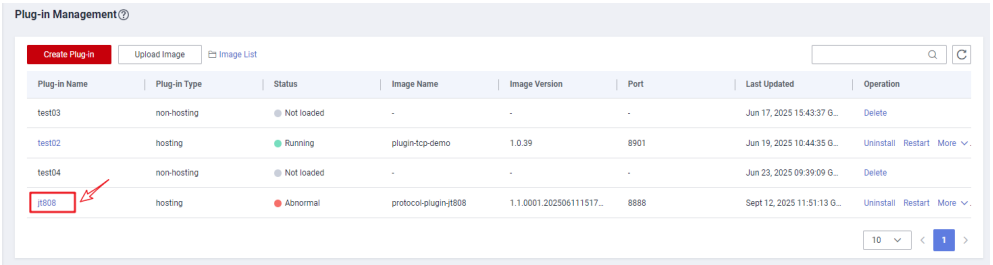
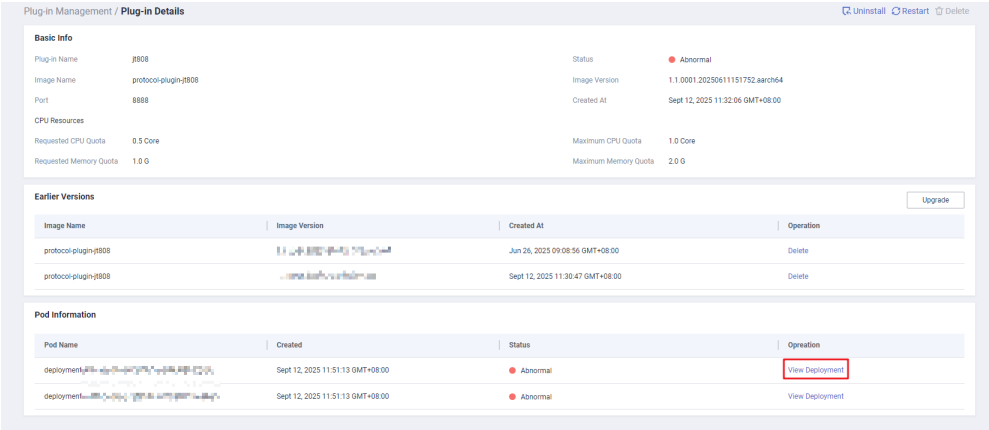


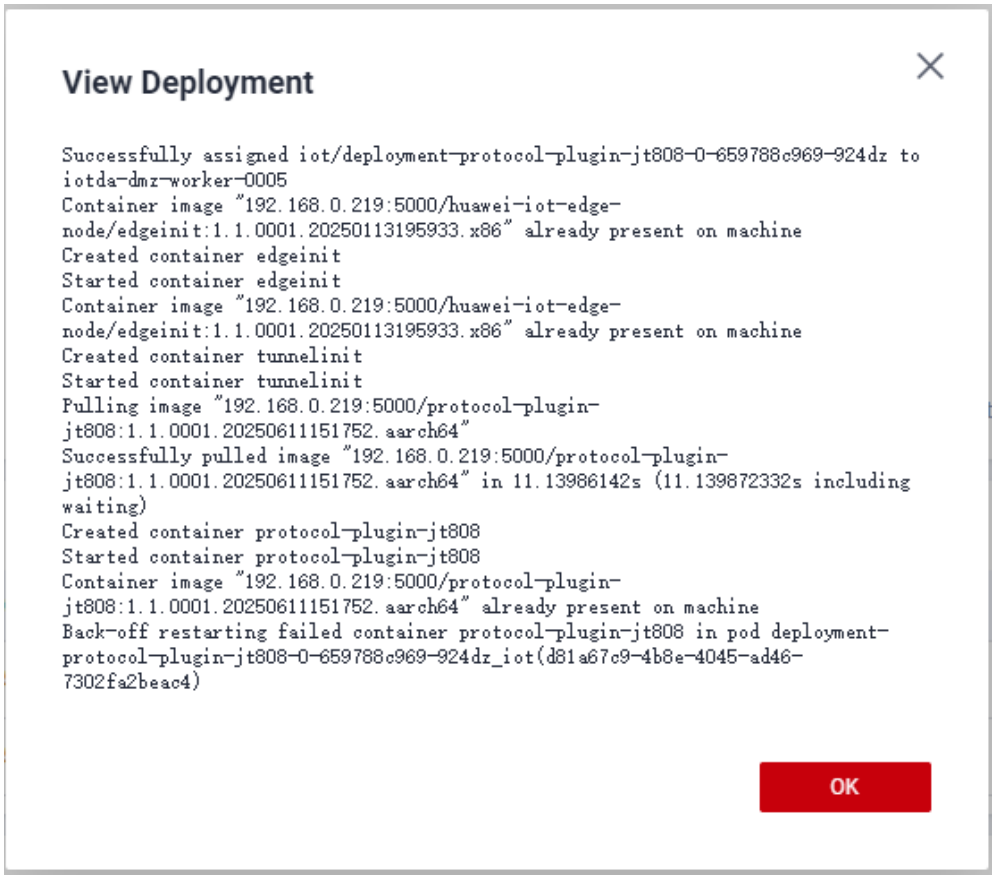
Figure 2-77 Plug-in details



1. Check deployment events.

   Click **View Deployment**. Kubernetes events of plug-in pod deployment are displayed.

   Figure 2-78 Deploying an event

   Deployment events can be used to troubleshoot plug-in deployment failures, including image pull failures and insufficient resources. (Note that deployment events are stored in the Kubernetes cluster for only 1 hour.)

----End

2.9 Message Communications

2.9.1 Overview

IoTDA supports bidirectional device communication. Devices can report data to IoTDA through APIs on the device side, and the platform can push the data to applications by using the subscription/push mechanism or forward the data to other services. Devices can also be remotely controlled by means of command delivery (using APIs or the IoTDA console).

Message communications are designed based on the product model. Devices can report properties and messages to the platform, and the platform can deliver commands, messages, and properties to the devices. Properties and commands are defined in the product model, whereas messages are not defined in the product model.

Table 2-13 Bidirectional communications

Data Type	Message Type	Differences	Similarities
Upstream data	Property reporting	Dependent on the product model. The properties reported must match those defined in the product model. You can check the latest device shadow on the device details page of the IoTDA console, push the data to the subscribed-to applications, and check historical data through IoT Analytics.	Both properties and messages can be reported to the platform by calling device-side APIs, and forwarded to other services using rules.
	Message reporting	Independent from the product model. The platform does not verify the message content. The latest reported device shadow data cannot be viewed on the device details page of the IoTDA console. Historical data cannot be viewed through IoT Analytics.	
Downstream data	Command delivery	Dependent on the product model. The commands delivered must match those defined in the product model. The command delivery is synchronous. (After a command is delivered, the platform waits for a response from the device. If no response is returned, the command delivery times out.) Commands can be delivered on the IoTDA console.	Commands, properties, and messages can be delivered by calling application-side APIs.

Data Type	Message Type	Differences	Similarities
	Property delivery (for modification purposes)	Dependent on the product model. The properties delivered must match those defined in the product model. Property delivery is synchronous. (After a property is delivered, the platform waits for a response from the device. If no response is returned, the modification times out.) The platform supports property configuration (device shadow) on the IoTDA console. Device properties can be modified through the device shadow to implement asynchronous mode.	
	Message delivery	Independent from the product model. The platform delivers messages to the device. Message delivery is asynchronous. (After a message is delivered, the platform does not need to wait for a response from the device.) Messages cannot be delivered from the IoTDA console.	

2.9.2 Data Reporting

Overview

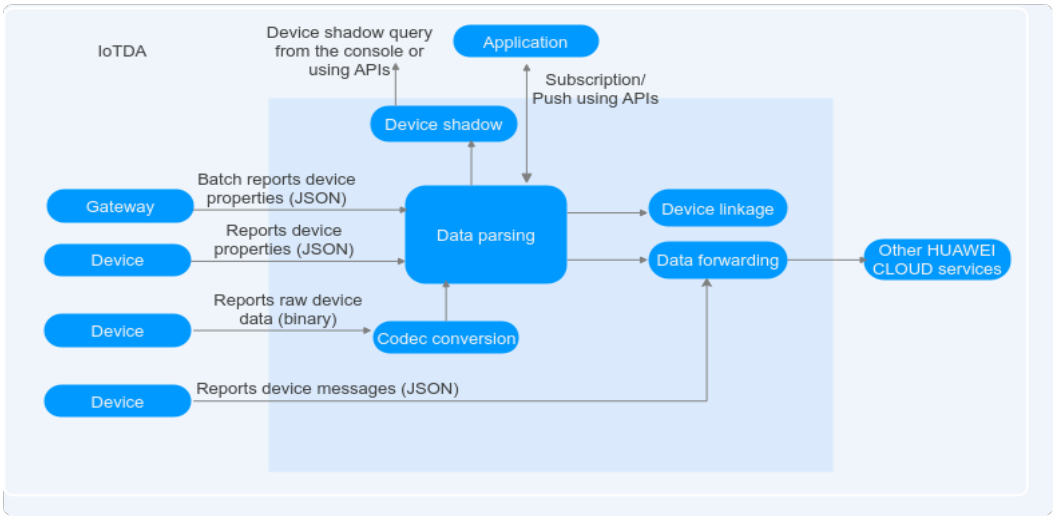
After being registered with IoTDA and powered on, a device can collect and report data based on the service logic. Data collection and reporting can be triggered by a schedule or by specific events. Devices can send data to IoTDA in the following ways:

- Reporting device messages: A device reports custom data to the platform through message reporting APIs. The platform does not parse or store the messages reported. Instead, it forwards the messages to other cloud services for storage and processing based on **data forwarding rules**. Then, the data is further processed through the consoles or APIs of the other services.
- Reporting raw device data (in binary or other non-standard formats): A device reports raw code streams to the platform. The platform uses codecs to parse the raw data into the standard JSON data defined in the product model and then performs subsequent processing. When MQTT is used, you can customize a topic to transparently transmit raw data.
- Reporting device properties: A device reports property data defined in the product model through property reporting APIs. The platform parses the data and then performs subsequent processing.
- Batch reporting device properties: A gateway reports data of a batch of devices to the platform at a time. The platform parses the data and then performs subsequent processing.

NOTE

Considering power consumption and bandwidth, CoAP devices can report only raw binary code streams. The platform uses codecs to convert the code streams into JSON data defined in the product model before performing subsequent processing.

Figure 2-79 Data reporting



2.9.3 Command Delivery

2.9.3.1 Mechanism

A product model defines commands that can be delivered to the devices. Applications can call IoTDA APIs to deliver commands to the devices to effectively manage these devices.

IoTDA supports synchronous and asynchronous command delivery.

Mechanism	Description	Application Scenario	Applicable Protocol
Synchronous command delivery	An application calls the synchronous command delivery API to deliver a command to a specified device for device control. The platform sends the command to the device and returns the command execution result in an HTTP request to the application. If the device does not respond, the platform returns a timeout message to the application.	Applicable to commands that must be executed in real time, for example, turning on a street lamp or closing a gas meter switch. Applications should determine the appropriate time to deliver a command.	MQTT

Mechanism	Description	Application Scenario	Applicable Protocol
Asynchronous command delivery	An application calls the asynchronous command delivery API to deliver a command to a specified device for device control. The platform sends the command to the device and asynchronously pushes the command execution result to the application. Asynchronous command delivery is classified into immediate delivery and delayed delivery. • In immediate delivery, the platform delivers commands to a device regardless of whether the device is online. If the device is offline or the device does not receive the command, the delivery fails. • In delayed delivery, IoTDA caches a command and delivers it to a device when the device goes online or reports data. If a device has multiple pending commands, the platform delivers the commands in sequence.	• Immediate delivery applies to scenarios with high real-time requirements. • Delayed delivery applies to commands that do not need to be executed immediately, for example, configuring water meter parameters.	CoAP

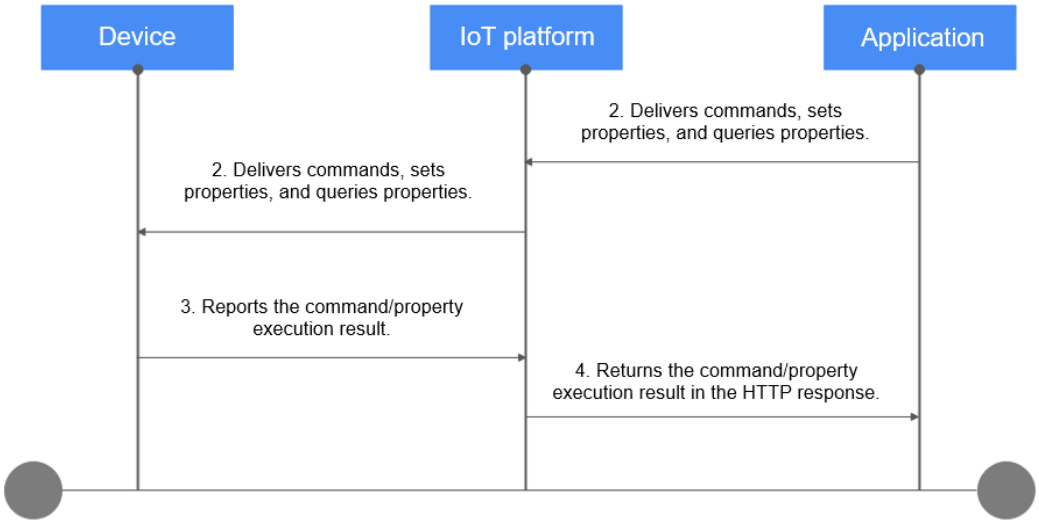
2.9.3.2 Command, Property, and Message Delivery for MQTT Devices

IoTDA supports command delivery, property setting, property query, and message delivery for MQTT devices. Messages can be delivered immediately or after a delay.

Command Delivery, Property Setting, and Property Query

The processes of command delivery, property setting, and property query are the same. After an application sends a command to a device, the application waits for a message carrying the command execution result from the device. This document uses command delivery as an example. For details on how to set and query

properties, see instructions of the APIs **Modifying Device Properties** and **Querying Device Properties**.



1. An application calls the API **Delivering a Device Command** to send a command to IoTDA. Example message:

```
POST https://{Endpoint}/v5/iot/{project_id}/devices/{device_id}/commands
Content-Type: application/json
X-Auth-Token: ********

{
  "service_id" : "WaterMeter",
  "command_name" : "ON_OFF",
  "paras" : {
    "value" : "ON"
  }
}
```

2. IoTDA sends the command to the device according to the protocol specifications.

   Example message:

```
Topic: $oc/devices/{device_id}/sys/commands/request_id={request_id}
Data format:
{
  "object_device_id": "{object_device_id}",
  "command_name": "ON_OFF",
  "service_id": "WaterMeter",
  "paras": {
    "value": "ON"
  }
}
```

3. After executing the command, the device returns the command execution result through the API **Delivering a Command**. Example message:

```
Topic: $oc/devices/{device_id}/sys/commands/response/request_id={request_id}
Data format:
{
    "result_code": 0,
    "response_name": "COMMAND_RESPONSE",
    "paras": {
        "result": "success"
    }
}
```

4. The platform synchronously sends a response to the HTTP command. Example message:

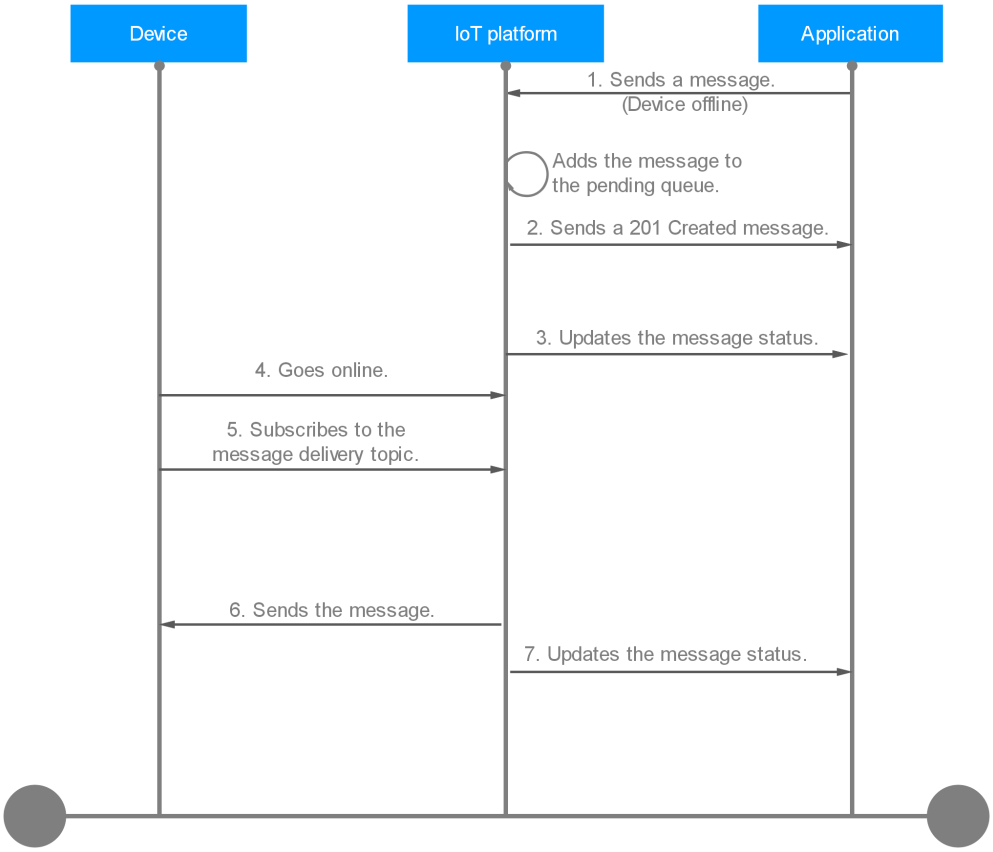

```
Status Code: 200 OK
Content-Type: application/json

{
  "command_id" : "b1224afb-e9f0-4916-8220-b6bab568e888",
  "response" : {
    "result_code" : 0,
    "response_name" : "COMMAND_RESPONSE",
    "paras" : {
      "result" : "success"
    }
  }
}
```

Message Delivery

MQTT device messages can be delivered immediately or after a delay. When a device is online, messages are delivered immediately. When a device is offline, messages are cached and delivered after the device goes online.

- **Delayed Delivery**

  This section describes the delayed delivery process of MQTT messages.



  a. An application calls the API **Delivering a Device Message** to send a message to IoTDA. Example message:

```
POST https://{Endpoint}/v5/iot/{project_id}/devices/{device_id}/messages
Content-Type: application/json
X-Auth-Token: ********

{
  "message_id": "99b32da9-cd17-4cdf-a286-f6e849cbc364",
  "name": "messageName",
  "message": "HelloWorld",
```



```
  "topic": "messageDown"
}
```

b. IoTDA sends a 201 **Created** message carrying the message status **PENDING** to the application.

c. IoTDA pushes the message result to the application through the API **Pushing a Device Message Status Change Notification**. Example message:

```
Topic: $oc/devices/{device_id}/sys/messages/down
Data format:
{
   "resource": "device.message.status",
   "event": "update",
   "notify_data": {
     "message_id": "string",
     "name": "string",
     "device_id": "string",
     "status": "PENDING",
     "timestamp": "string"
    }
}
```

d. The device goes online.

e. The device subscribes to the message delivery topic, which is used to receive messages. For details on the subscribed topic, see **6**.

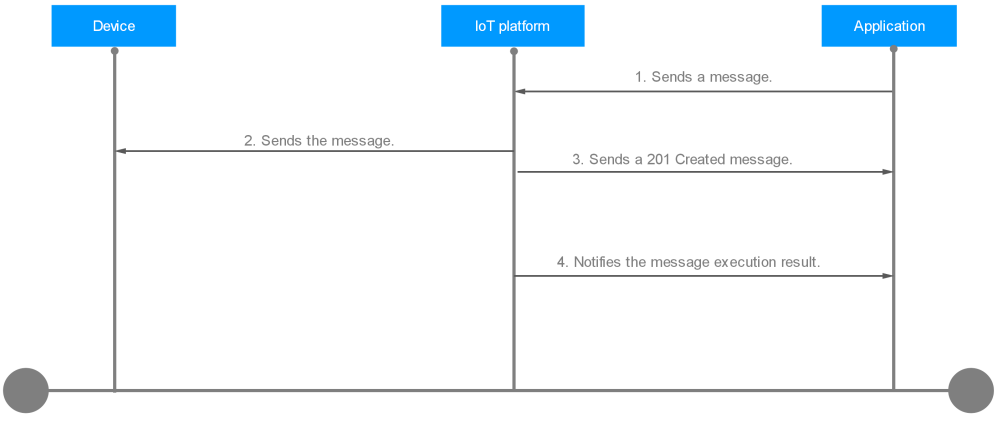
f. IoTDA delivers messages to devices based on protocol specifications. Example message:

```
Topic: $oc/devices/{device_id}/sys/messages/down
Data format:
{
   "object_device_id": "{object_device_id}",
   "name": "name",
   "id": "id",
   "content": "hello"
}
```

g. The platform pushes the final result of the message to the application. For details on the message structure, see **3**.

- **Immediate Delivery**

  This section describes the immediate delivery process of MQTT messages.



a. An application calls the API **Delivering a Device Message** to send a message to IoTDA. Example message:

```
POST https://{Endpoint}/v5/iot/{project_id}/devices/{device_id}/messages
Content-Type: application/json
X-Auth-Token: ********

{
```



```
  "message_id": "99b32da9-cd17-4cdf-a286-f6e849cbc364",
  "name": "messageName",
  "message": "HelloWorld",
  "topic": "messageDown"
}
```

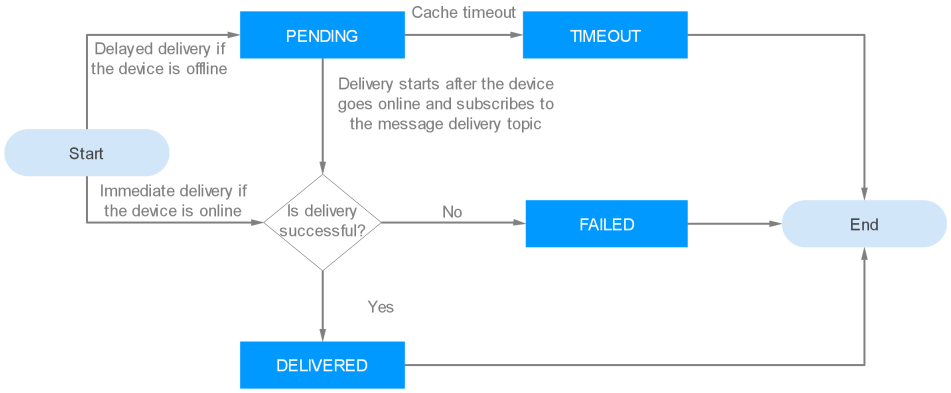
b. IoTDA delivers messages to devices based on protocol specifications. Example message:

```
Topic: $oc/devices/{device_id}/sys/messages/down
Data format:
{
   "object_device_id": "{object_device_id}",
   "name": "name",
   "id": "id",
   "content": "hello"
}
```

c. IoTDA pushes the message result to the application through the API **Pushing a Device Message Status Change Notification**. Example message:

```
Topic: $oc/devices/{device_id}/sys/messages/down
Data format:
{
   "resource": "device.message.status",
   "event": "update",
   "notify_data": {
     "message_id": "string",
     "name": "string",
     "device_id": "string",
     "status": "string",
     "timestamp": "string"
    }
 }
```

Message Delivery Status



Status	Description
PENDING	If an MQTT device is offline, IoTDA caches the message. In this case, the task status is **PENDING**.
TIMEOUT	If IoTDA does not deliver the message in the pending status within one day, the task status changes to **TIMEOUT**.
DELIVERED	After IoTDA sends the message to the device, the task status changes to **DELIVERED**.

Status	Description
FAILED	If IoTDA fails to send a message to the device, the task status changes to **FAILED**.

Synchronous Command Delivery to an MQTT Device

The platform supports command delivery to an MQTT device by calling the API **Delivering a Device Command** or creating a command delivery task on the console. This section describes how to create a command delivery task on the console.

Step 1 Log in to the IoTDA console.

Step 2 In the left navigation pane, choose **Devices > All Devices**. In the device list, click **View** in the row of a device to access its details.

Step 3 Click the **Commands** tab, and click **Deliver Command** in the **Synchronous Command Delivery** area. In the dialog box that is displayed, enter your information and click **Yes**.

NOTE

- The commands can be delivered only after being defined in the product model.
- MQTT devices support only synchronous command delivery. Devices using CoAP support only asynchronous command delivery.

----End

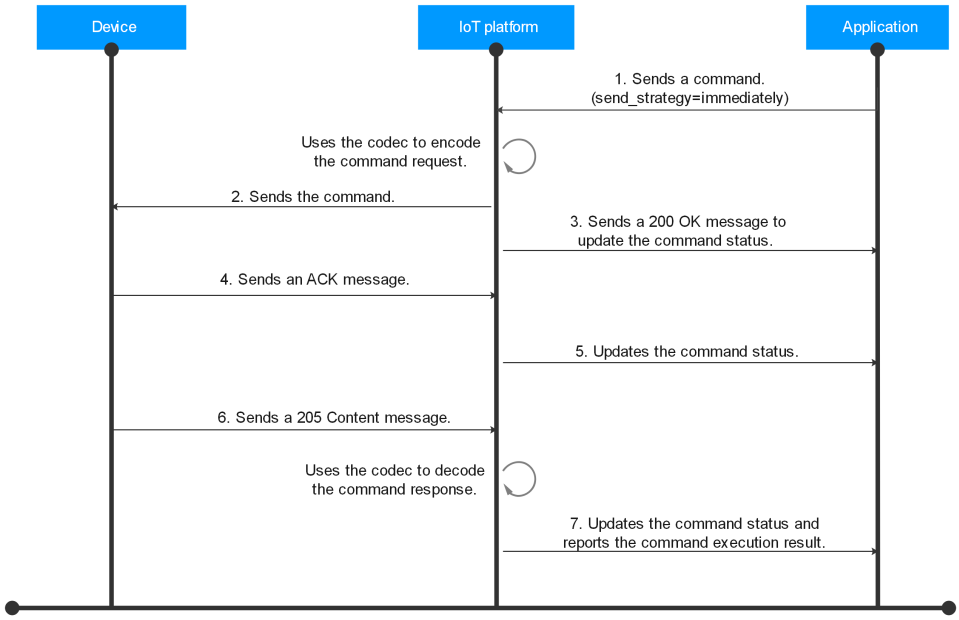
2.9.3.3 Command Delivery for Devices Using CoAP

For devices using CoAP, IoTDA provides two delivery mechanisms: immediate delivery and delayed delivery. The parameter **send_strategy** in the command delivered from an application to IoTDA specifies the delivery mechanism.

- If **send_strategy** is **immediately**, the command is delivered immediately.
- If **send_strategy** is **delay**, the command is cached before being delivered.

Immediate Delivery

This section describes the process of immediate command delivery.



1. An application calls the API **Delivering an Asynchronous Command** to send a command to IoTDA. The **send_strategy** parameter in the command request is set to **immediately**. Example message:

```
POST https://{endpoint}/v5/iot/{project_id}/devices/{device_id}/async-commands
Content-Type: application/json
X-Auth-Token: ********

{
  "service_id" : "WaterMeter",
  "command_name" : "ON_OFF",
  "paras" : {
    "value" : "ON"
  },
   "expire_time": 0,
   "send_strategy": immediately
}
```

2. IoTDA uses the codec to encode the command request, and sends the command through the **Execute** operation of the APIs for device management and service implementation defined in the CoAP protocol. The message body is in binary format.
3. IoTDA sends a 200 **OK** message carrying the command status **SENT** to the application. (If the device is offline or the device does not receive the command, the delivery fails and the command status is **FAILED**.)
4. The device returns an ACK message after receiving the command.
5. If the application has subscribed to command status change notifications, IoTDA pushes a message to the application by calling the API **Pushing a Command Status Change Notification**. The command status carried in the message is **DELIVERED**. Example message:

```
Method: POST
request:
Body:
{
    "resource": "device.command.status",
    "event": "update",
    "event_time": "20200811T080745Z",
```



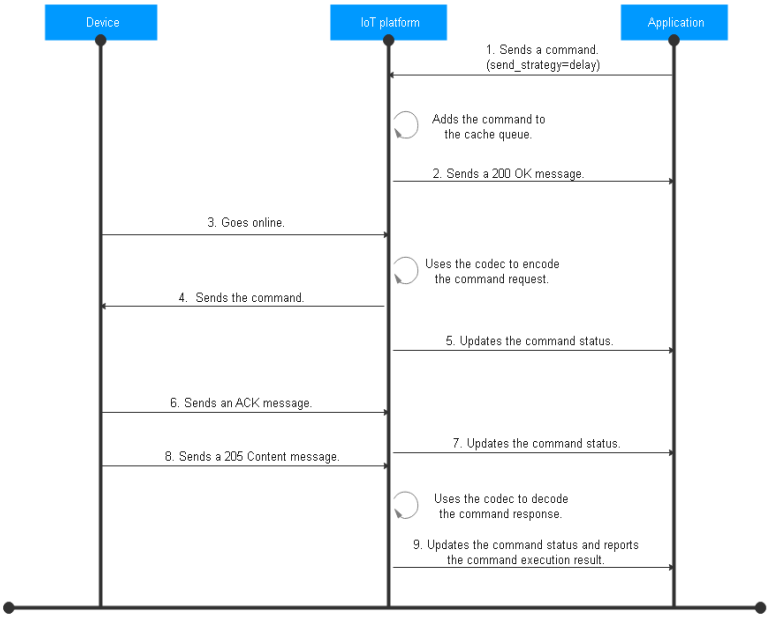
```
    "notify_data": {
        "header": {
            "app_id": "8d4a34e5363a49bfa809c6bd788e6ffa",
            "device_id": "5f111a5a29c62ac7edc88828_test0001",
            "node_id": "test0001",
            "product_id": "5f111a5a29c62ac7edc88828",
            "gateway_id": "5f111a5a29c62ac7edc88828_test0001",
            "tags": []
        },
        "body": {
            "command_id": "49ca40af-7e14-4f7b-b97b-78cdd347a6b9",
            "created_time": "20200811T080738Z",
            "sent_time": "20200811T080738Z",
            "delivered_time": "20200811T080745Z",
            "response_time": "",
            "status": "DELIVERED",
            "result": null
        }
    }
}
```

6. After the command is executed, the device returns the command execution result in a 205 Content message.
7. If the application has subscribed to command status change notifications, IoTDA uses the codec to decode the command response and sends a push message to the application by calling the API **Pushing a Command Status Change Notification**. The command status carried in the message is **SUCCESSFUL**. Example message:

```
Method: POST
request:
Body:
{
    "resource": "device.command.status",
    "event": "update",
    "event_time": "20200811T080745Z",
    "notify_data": {
        "header": {
            "app_id": "8d4a34e5363a49bfa809c6bd788e6ffa",
            "device_id": "5f111a5a29c62ac7edc88828_test0001",
            "node_id": "test0001",
            "product_id": "5f111a5a29c62ac7edc88828",
            "gateway_id": "5f111a5a29c62ac7edc88828_test0001",
            "tags": []
        },
        "body": {
            "command_id": "49ca40af-7e14-4f7b-b97b-78cdd347a6b9",
            "created_time": "20200811T080738Z",
            "sent_time": "20200811T080738Z",
            "delivered_time": "20200811T080745Z",
            "response_time": "20200811T081745Z",
            "status": "SUCCESSFUL",
            "result": {
                "resultCode":"SUCCESSFUL",
                "resultDetail": {
                    "value": "ON"
                }
            }
        }
    }
}
```

Delayed Delivery

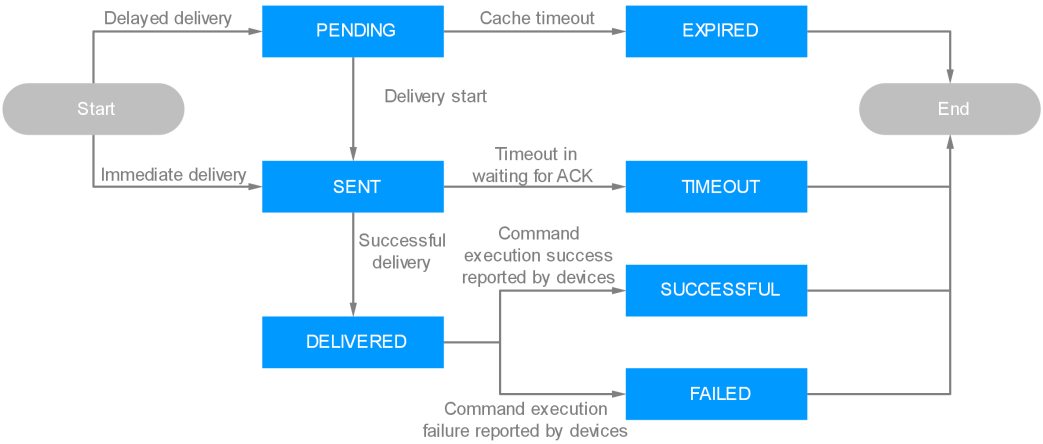
This section describes the process of delayed command delivery.



1. An application calls the API **Delivering an Asynchronous Command** to send a command to IoTDA. The **send_strategy** parameter in the command request is set to **delay**.
2. IoTDA adds the command to the cache queue and reports a 200 OK message. The command status is **PENDING**.
3. The device goes online or reports data to the platform.
4. IoTDA uses the codec to encode the command request and sends the command to the device according to the protocol specifications.
5. If the application has subscribed to command status change notifications, IoTDA pushes a message to the application by calling the API **Pushing a Command Status Change Notification**. The command status carried in the message is **SENT**.
6. The subsequent flow is the same as **4** to **7** described in the immediate delivery scenario.

Command Execution Status

The figure below illustrates the command execution status and the table below describes the status change mechanism.



Status	Description
PENDING	• For a device using CoAP in delayed delivery mode, IoTDA caches a command if the device has not reported data. The command status is **PENDING**. • This status does not exist for devices using CoAP in immediate delivery mode.
EXPIRED	• For a device using CoAP in delayed delivery mode, if IoTDA does not deliver a command to the device within a specified time, the command status is **EXPIRED**. The expiration time is subject to the value of **expireTime** carried in the command request. If **expireTime** is not carried, the default value (24 hours) is used. • This status does not exist for devices using CoAP in immediate delivery mode.
SENT	• For a device using CoAP in delayed delivery mode, IoTDA sends a cached command when receiving data reported by the device. In this case, the command status changes from **PENDING** to **SENT**. • For a device using CoAP in immediate delivery mode, if the device is online when the platform delivers a command, the command status is **SENT**.
TIMEOUT	If IoTDA does not receive a response within 180 seconds after delivering a command to a device using CoAP, the command status is **TIMEOUT**.
DELIVERED	If IoTDA receives a response from a device, the command status is **DELIVERED**.
SUCCESSFUL	If IoTDA receives a result indicating that the command is executed, the command status is **SUCCESSFUL**.

Status	Description
FAILED	• If IoTDA receives a result indicating that the command execution failed, the command status is **FAILED**. • For a device using CoAP in immediate delivery mode, if the device is offline when the platform delivers a command, the command status is **FAILED**.

Asynchronous Command Delivery

The platform supports command delivery to a device using CoAP by calling the API **Delivering a Device Command** or creating a command delivery task on the console. This section describes how to create a command delivery task on the console.

Step 1 Log in to the IoTDA console.

Step 2 In the left navigation pane, choose **Devices > All Devices**. In the device list, click **View** in the row of a device to access its details.

Step 3 Click the **Commands** tab, and click **Deliver Command** in the **Asynchronous Command Delivery** area. In the dialog box that is displayed, enter your information and click **Yes**.

NOTE

- The commands can be delivered only after being defined in the product model.
- MQTT devices support only synchronous command delivery. Devices using CoAP support only asynchronous command delivery.

----End

2.9.4 Custom Topic Communications

Overview

The platform uses topics to communicate with devices over MQTT. There are custom topics and system topics. System topics are basic communication topics preconfigured on the platform. You can also customize topics on the platform based on service requirements.

IoTDA enables transparent data transmission of MQTT custom topics to simplify the process of migrating MQTT devices from other platforms to IoTDA.

- This feature allows devices that comply with MQTT specifications to be migrated to IoTDA without any modification. In addition, the data pushed by IoTDA to upper-layer applications is compatible with the original format of the devices, which reduces the adaptation and modification workload of devices and upper-layer applications to be quickly migrated to IoTDA.
- This feature enables devices to quickly connect to native MQTT devices without a large amount of adaptation and modification on the original devices and applications. In addition, codecs are optional. You can continue to develop encoding and decoding scripts to interact with upper-layer

applications based on the IoTDA model, or choose to transparently transmit data.

Scenarios

- Devices publish messages to a custom topic, from which an application receives the messages using the **Data Forwarding** function.
- An application calls the API for delivering a message to a device to publish messages to a specified custom topic. Devices subscribe to this topic to receive messages from the server.

Category

Table 2-14 Topic classification

Category	Description
System topic	The platform predefines topics for communications with devices. For details of the topic list and functions, see "API Reference on the Device Side" > "Topics" in the *API Reference*.
Custom topic	Devices and the platform can communicate based on custom topics. Types of custom topics: • Topics defined in the product are prefixed with **$oc/devices/*{device_id}*/user/**. Therefore, these topics are also called custom topics with fixed prefix. During message reporting or delivery, the platform checks whether the topic is defined in the product. Undefined topics will be rejected by the platform. • Topics that do not start with **$oc**, for example, **/aircondition/data/up**. These topics enable upstream and downstream message communications based on MQTT rules. The platform does not verify the topic permission.

Constraints

- You can define a maximum of 50 custom topics for a product model by default.
- Custom topics are only available for message communications.
- Maximum length of a custom MQTT topic: 128 bytes.

Adding a Custom Topic

1. Log in to the IoTDA console.
2. Choose **Products** in the left navigation pane.
3. Select an MQTT product. On the product details page, click the **Topic Management** tab, select **Custom Topics**, and click **Add Fixed Prefix Topic**.
4. In the dialog box displayed, select device operation permissions and enter the topic name.

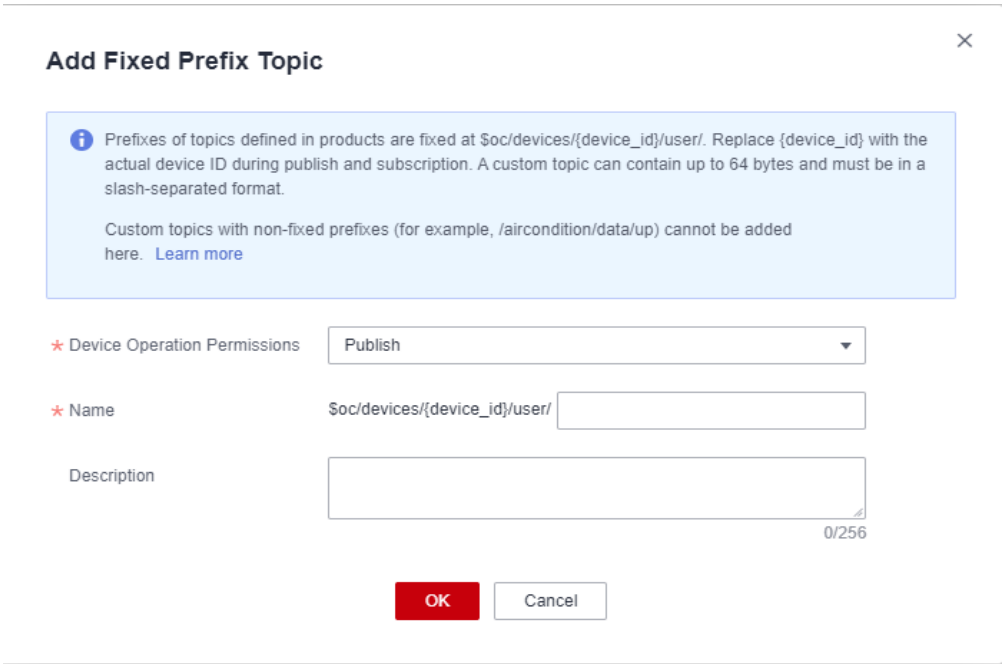


Table 2-15

Parameter	Description
Device Operation Permissions	• **Publish**: Devices can report messages using this topic. A topic is carried in a device message during data transfer for better classification. • **Subscribe**: Applications can specify a topic to deliver messages to devices. • **Publish & Subscribe**: Devices can report and receive messages using this topic.
Name	The topic prefix is fixed at **$oc/devices/***{device_id}***/user/**. Replace *{device_id}* with the actual device ID during publishing and subscription. A custom topic must be in a slash-separated format. Enter 1 to 64 characters. Use only digits, letters, underscores (_), and slashes (/). The slash cannot be consecutive or be the end. Notes: • Custom topics do not support custom variables. For example, *{type}* in **$oc/devices/***{device_id}***/user/setting/***{type}* is a variable and is not supported. • Currently, custom topics do not support wildcard characters and cannot contain plus signs (+) or number signs (#)
Description	Description of the topic.

5. Click **OK**.

   After the topic is added, you can modify or delete it in the custom topic list.

Transparent Data Transmission Processes

The following figure shows the transparent data transmission process. Compared with the original process, the difference lies in the message format during transmission.

Figure 2-80 Custom Topic Data Transparent Transmission

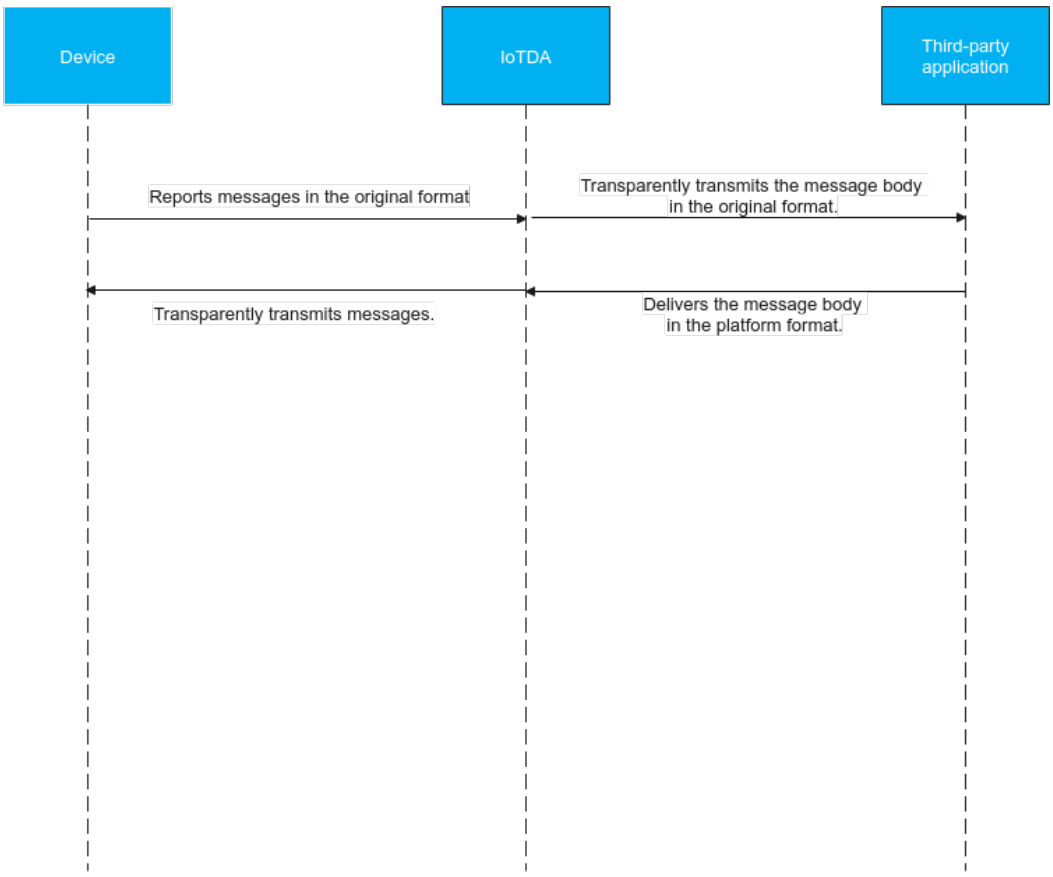
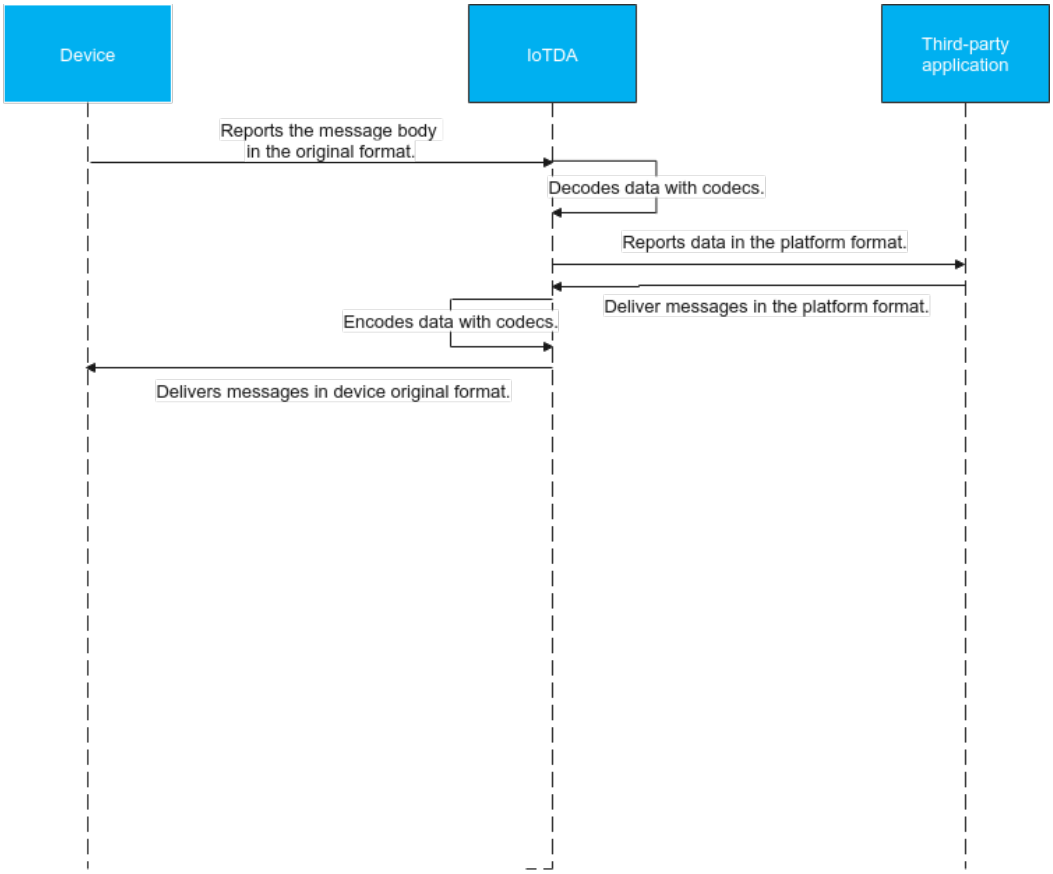
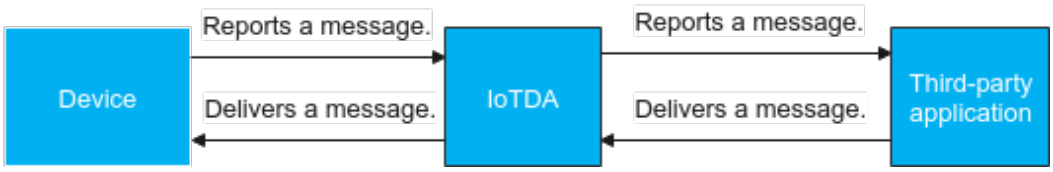


Figure 2-81 Original process



Transparent Transmission Modes

The following shows the comparison between SDK integration and custom topic data transparent transmission.



SDK integration

- Devices must be integrated with the IoT Device SDK or reconstructed based on the IoT southbound MQTT API specifications of IoTDA.
- Codecs are optional. If the device and IoTDA exchange messages based on the product model, the codec is not required. Otherwise, a codec needs to be developed to convert the upstream messages of the device into the product model messages that can be understood by IoTDA, and convert the product model message delivered by IoTDA into a message that can be understood by the device.
- Upper-layer applications need to be integrated and developed based on the data push format of IoTDA.

Custom topic data transparent transmission

- Devices can access the platform in native MQTT mode without integrating with the IoT Device SDK. For details about the authentication mode, see "API Reference on the Device Side" > "Device Connection Authentication" in *API Reference*.
- Codec development is optional. Messages can be transparently transmitted. Therefore, codec script development and upper-layer application adaptation are not required.

Message Reporting with Data Transparent Transmission

For details about how to develop the codec and the encoding and decoding processes, see "Developing a Codec" in *Developer Guide*.

Table 2-16 Example

Scenario	Data Type	Data Reported by the Device	Data Received by the Platform

Check the format of the reported topic. If the topic starts with **$oc**, the message is a system message and is directly reported. The reported data can be of the **Object** or **String** format.	String	topic:$oc/devices/{device_id}/sys/messages/up
		{
		"object_device_id": "{object_device_id}",
		"name": "name",
		"id": "id",
		"content": "hello"
		}
			{
			"topic": "$oc/devices/{device_id}/sys/messages/up",
			"content": {
			"object_device_id": "{object_device_id}",
			"name": "name",
			"id": "id",
			"content": "hello"
			}
			}

Scenario	Data Type	Data Reported by the Device	Data Received by the Platform

	Object	topic:$oc/devices/{device_id}/sys/messages/up
		{
		"object_device_id": "{object_device_id}",
		"name": "name",
		"id": "id",
		"content": {
		"name": "huhu"
		}
		}
			{
			"topic": "$oc/devices/{device_id}/sys/messages/up",
			"content": {
			"object_device_id": "{object_device_id}",
			"name": "name",
			"id": "id",
			"content": {
			"name": "huhu"
			}
			}
			}

For topics whose names do not start with **$oc**, obtain the value of **data_format** in the product information. Data is converted based on the value of **data_format**.	For JSON format, convert data to the JSON format.	topic: /messages/custom/up
		{
		"action": "test"
		}
			{
			"topic": "/messages/custom/up",
			"content": {
			"action": "test"
			}
			}

	For binary format, data needs to be Base64-encrypted before being pushed.	topic: /messages/custom/up
		"data"
			{
			"topic": "/messages/custom/up",
			"content": "base64(data)"
			}

Message Delivery with Data Transparent Transmission

For details about how to develop the codec and the encoding and decoding processes, see "Developing a Codec" in *Developer Guide*.

Key Fields (DeviceMessage)	Data Type	Field Description
topic	String	Topic in the standard format of the platform, which starts with **$oc/devices/*{device_id}*/**.
full_topic	String	Custom topic, which is corresponding to the **topic_full_name** field transferred from the platform.
message_id	String	Message ID, which is globally unique.
base64_data	String	Custom topic, which can carry the Base64-encoded message body.
data	String	Non-encoded message body that can be carried by a custom topic or a topic in the platform format.

NOTE

- You can select either **topic** (topic in the platform format) or **full_topic** (custom topic). If both fields contain data, the system reports an error.
- If you select the custom topic, **base64_data** and **data** are available. If you select the topic in the platform format, only **data** can be carried.

Table 2-17 Example

Scenario	Message Delivered by the Platform	Message Received by the Device

If **encoding** is set to **base64**, the message is delivered as the payload.	{
	"message_id": "99b32da9-cd17-4cdf-a286-f6e849cbc364",
	"name": "messageName",
	"message": "dGhpcyBpcyBhIGV4YW1wbGU",
	"encoding": "base64",
	"payload_format": "standard",
	"topic_full_name": "/device/custom/down"
	}
		"this is an example" (*original information after the message is decoded using Base64*)

Scenario	Message Delivered by the Platform	Message Received by the Device

When **encoding** is set to **none** and **payload** is set to **raw**, the message is delivered as the payload.	{
	"message_id": "99b32da9-cd17-4cdf-a286-f6e849cbc364",
	"name": "messageName",
	"message":{
	"name":"test",
	"action":"move"
	},
	"encoding": "none",
	"payload_format": "raw",
	"topic_full_name": "/device/custom/down"
	}
		{
		"name":"test",
		"action":"move"
		}

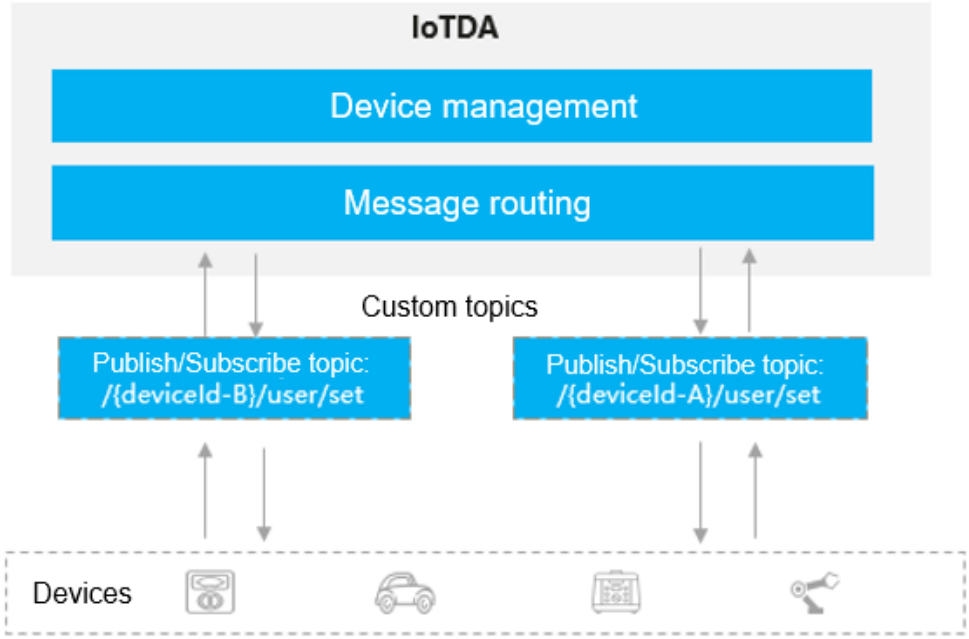
When **encoding** is set to **none** and **payload** is set to **standard**, the message is delivered in the standard format of IoTDA.	{
	"message_id": "99b32da9-cd17-4cdf-a286-f6e849cbc364",
	"name": "messageName",
	"message":{
	"name":"test",
	"action":"move"
	},
	}
		{
		"object_device_id": "12345678901",
		"name": "reboot",
		"id": "709b334b-568d-4d5a-960a-1666d775f2c6",
		"content": {
		"name": "test",
		"action":"move"
		}
		}

2.9.5 M2M Communications

Overview

IoTDA supports MQTT-based machine-to-machine (M2M) communications. The platform processes the connection and communication requests from devices, so you can focus on service implementation.

Figure 2-82 Service flow



NOTE

- By default, you do not have the permission to enable the inter-device message communications. Contact R&D engineers to configure the permission.

Scenarios

- Instant messaging scenario where a sender and recipient communicate with each other.
- Smart home scenario where messages are exchanged between mobile apps and smart devices.

Constraints

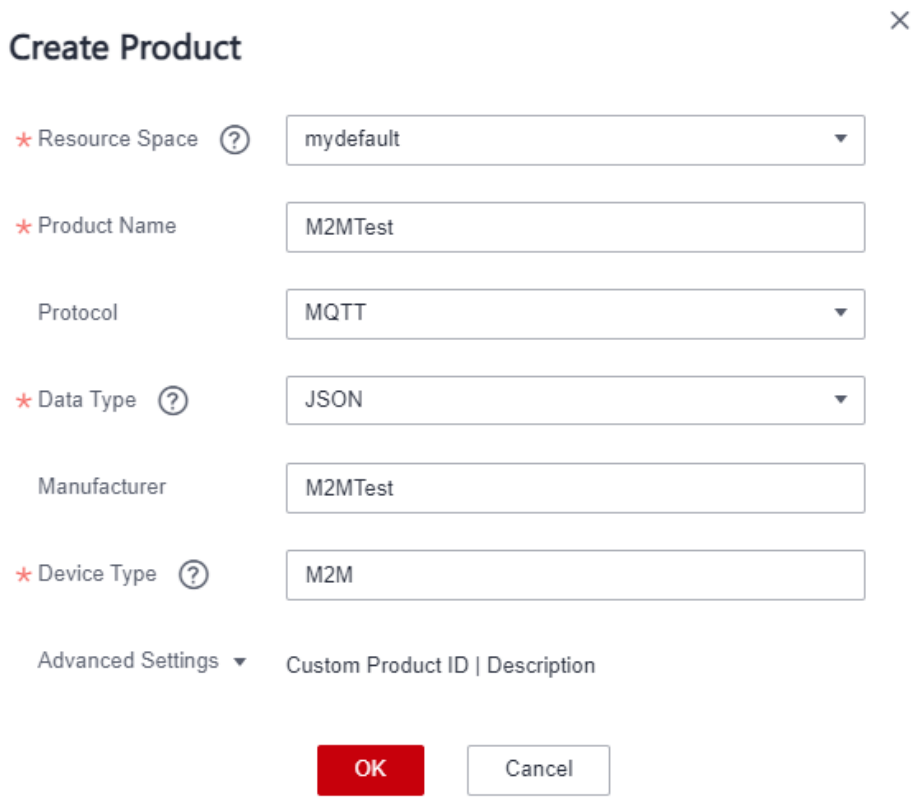
1. Only one-to-one communications are supported.
2. Max. message size: 64 KB.
3. Max. M2M message reporting TPS: 500/s.
4. M2M messages are not cached. If the recipient device is offline, the messages will be lost.

Procedure

Step 1 To use the M2M communications, you need to configure a topic in the product and assign the publish and subscribe permissions to the topic. The procedure is as follows:

1. Log in to the IoTDA console.
2. In the navigation pane, choose **Products**. Select **MQTT** for **Protocol**.

Figure 2-83 Creating a product



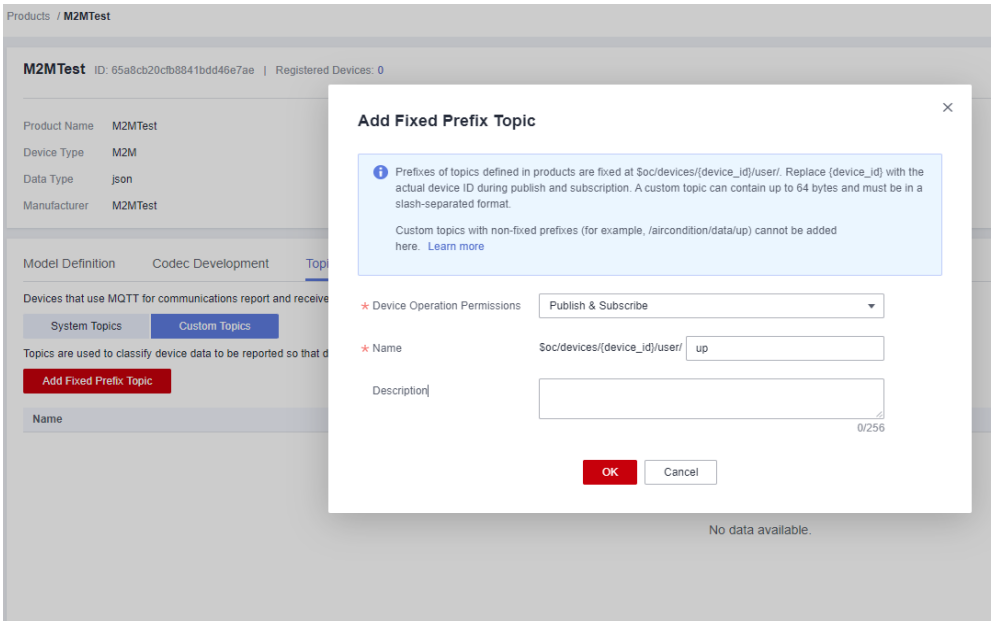
CAUTION

Select **JSON** for **Data Type** if the inter-device communications use JSON data.

Select **Binary** for **Data Type** if the inter-device communications do not use JSON data.

3. Create a custom topic in the product, for example, **$oc/devices/***{device_id}*/**user/up**. The topic must have publish and subscribe permissions.

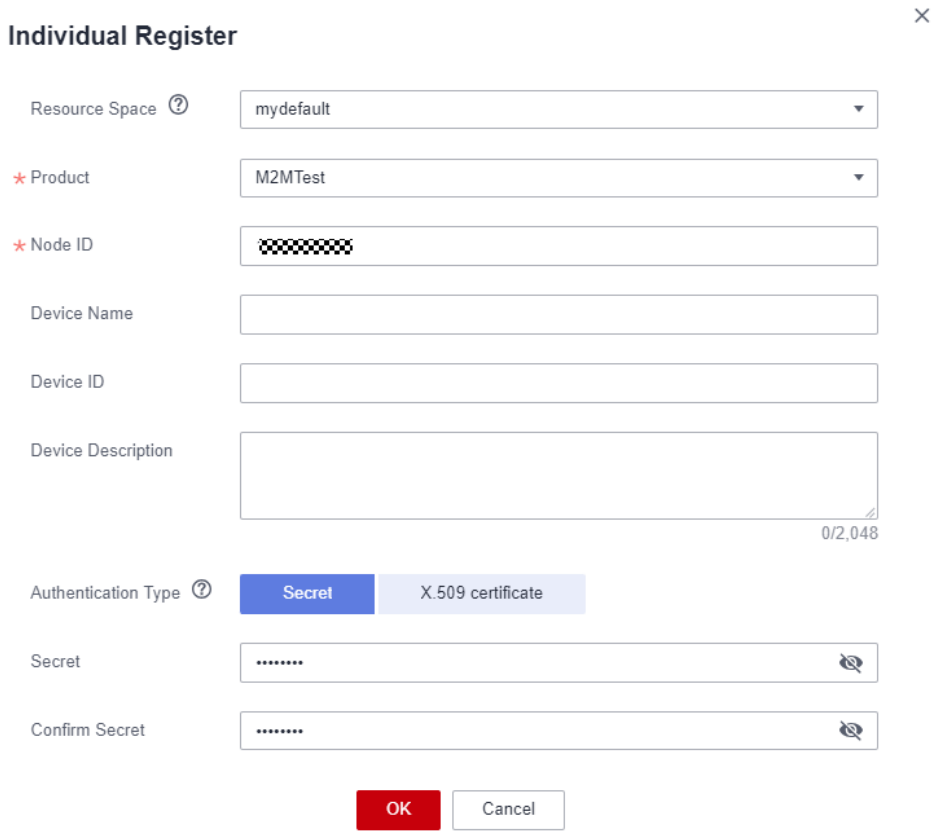
Figure 2-84 Custom topic



Step 2 After configuring a topic, you can implement M2M communications based on your service scenario. The following uses MQTT.fx as an example to describe how to implement M2M communications.

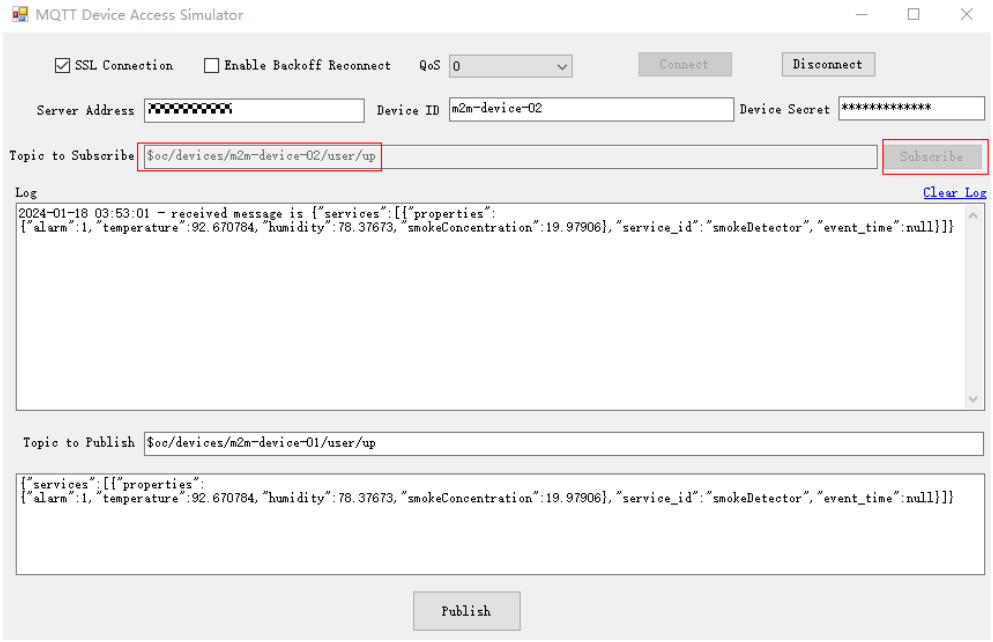
1. Register device A and device B under the product created in **1**.

Figure 2-85 Registering a device



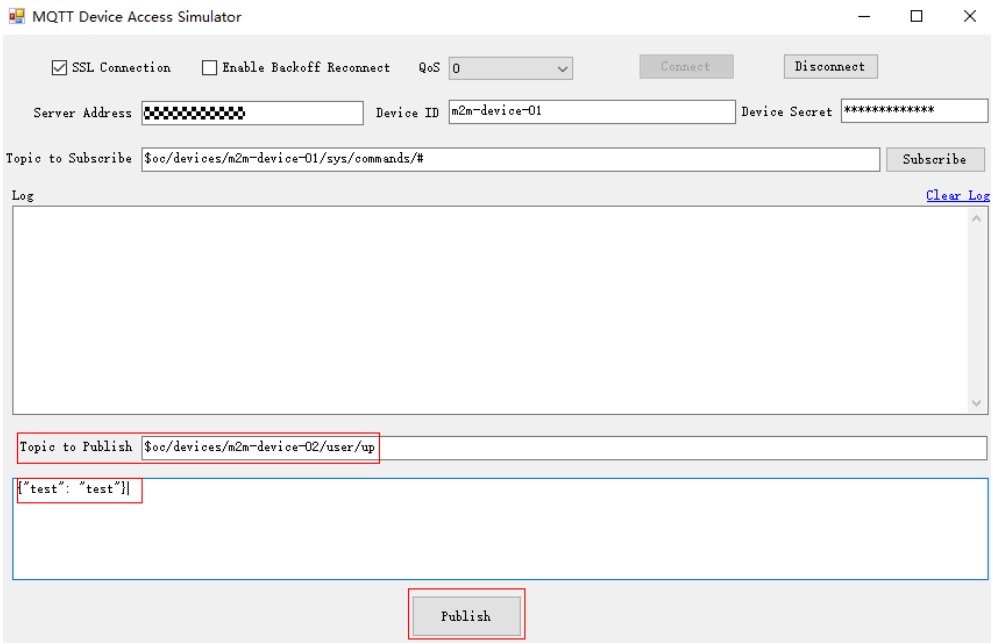
2. Start two MQTT simulators to simulate devices A and B, respectively. For details about how to use the MQTT simulator, see "Development on the Device Side" > "Accessing Using MQTT Demos" > "Using the MQTT Simulator for Debugging" in the *Developer Guide*.
3. Use device B to subscribe to the topic **$oc/devices/***{device_id}***/user/up**. Replace *{device_id}* with the device ID of device B.

Figure 2-86 Device B subscribing to topics



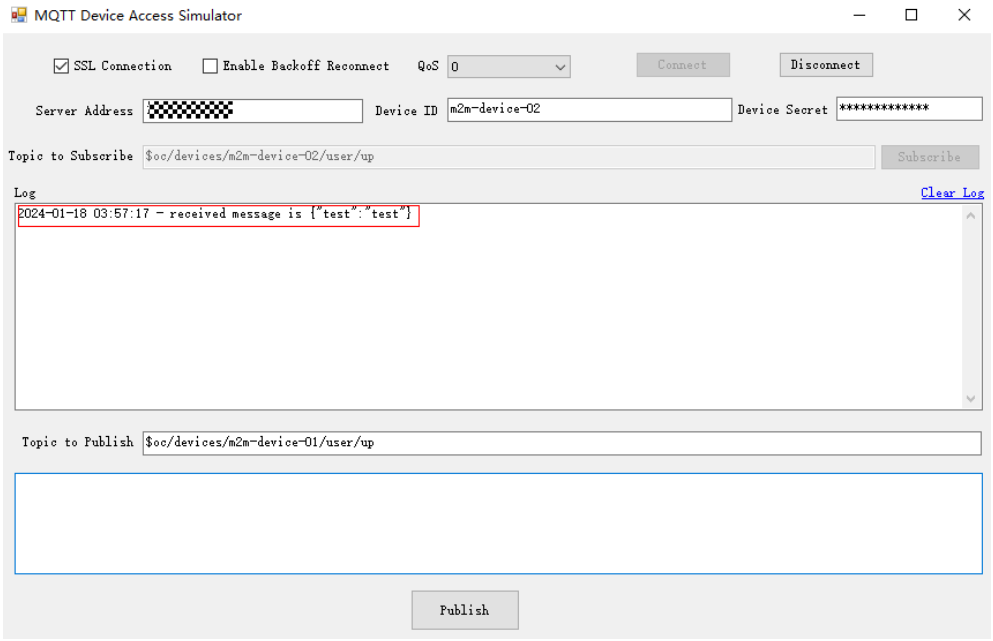
4. Use device A to send a message to device B. On the MQTT simulator of device A, enter topic **$oc/devices/***{device_id}***/user/up** (replace *{device_id}* with the device ID of device B). Enter the message to be sent in the content text box and click **Publish**.

Figure 2-87 Device A sending a message



Device B receives the message from device A, as shown in the following figure.

Figure 2-88 Device B receiving a message



----End

2.10 Subscription/Push

2.10.1 Overview

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

For details about the subscription modes supported by IoTDA, see **Data Forwarding**.

Subscribing to Data

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

- For details about how to configure subscriptions on the console, see **Data Forwarding**.

Format of Pushed Data

After the data subscription is successful, IoTDA pushes the data to the application. For details about the data format example, see "Data Transfer APIs".

NOTE

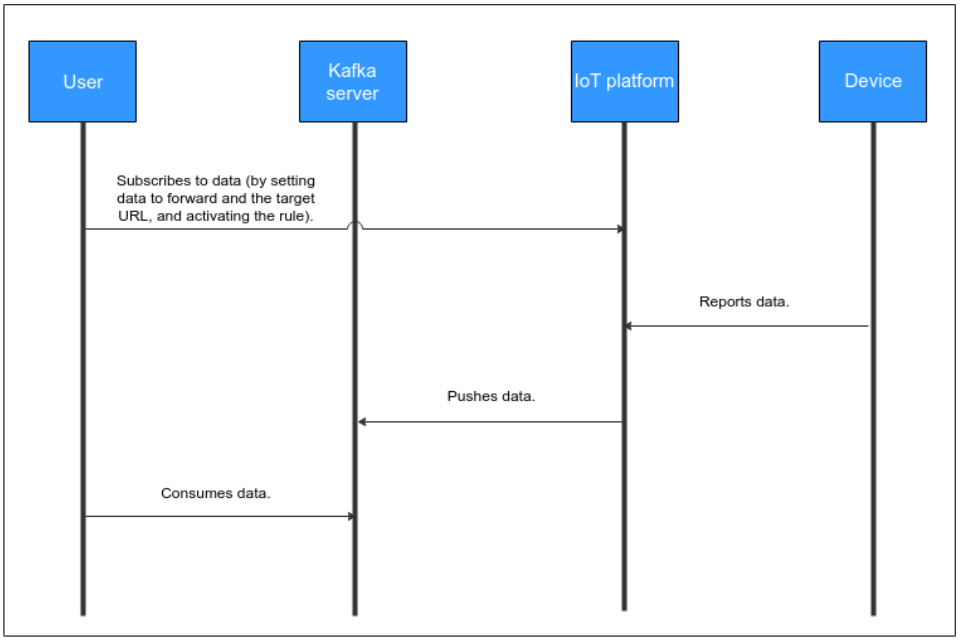
In the HTTP message header, the value of **Content-Type** is **application/json**, and the character set is **UTF-8**.

2.10.2 Kafka Subscription/Push

Subscription: Distributed Message Service (DMS) for Kafka, ROMA Connect, and Third-party message queue (Kafka) are based on the Kafka protocol. By creating a subscription task on the IoTDA console, or calling IoTDA APIs to configure and activate rules, you can obtain changed device service details (such as device lifecycle management, device data reporting, device message status, and device status) and management details (software/firmware upgrade status and result) from IoTDA. The Kafka message channel must be specified during subscription creation.

Push: After a subscription is created, IoTDA pushes the corresponding change to the specified Kafka message service based on the type of data subscribed. If an application does not subscribe to a specific type of data notification, IoTDA does not push the data to the application even if the data has changed. You can use the Kafka client to establish a connection with IoTDA to receive data.

Figure 2-89 Kafka subscription/push



Push mechanism: After IoTDA pushes messages to the Kafka server, you need to consume messages on the Kafka server. Otherwise, data will be stacked on the Kafka server.

Subscribing to Data

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

- For details on how to configure Kafka subscriptions on the console, see **Configuring Kafka Subscription**.

Format of Pushed Data

After the data subscription is successful, IoTDA pushes the data to the application. For details about the data format example, see "Data Transfer APIs".

Configuring Kafka Subscription

This section describes how to configure Kafka subscription on the IoTDA console.

Step 1 Log in to the IoTDA console.

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

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

Parameter	Description
Rule Name	Specify the name of a rule to create.
Description	Describe the rule.
Data Source	• **Device property**: A property value reported by a device in a resource space will be forwarded. • **Device message**: A message reported by a device in a resource space will be forwarded. • **Device message status**: The status of device messages exchanged between the device and platform will be forwarded. • **Device status**: The status change of a directly connected device in a resource space will be forwarded. • **Device event**: Only events you defined in the product will be forwarded. • **Batch task**: The batch task status will be forwarded. • **Product**: Product information, such as product addition, deletion, and update, will be forwarded. • **Device**: Device information, such as device addition, deletion, and update, will be forwarded. • **Device alarm**: Device alarm information, such as device alarm generation and clearance, will be forwarded. • **Asynchronous command status of the device**: The command status change of the device will be forwarded.
Trigger	After the data source is selected, the platform automatically matches the trigger event.
Resource Space	You can select a single resource space or all resource spaces.
SQL Filter Statements	You can edit the SQL statements for processing message data and set the data filtering statements. Click **Edit SQL** to edit the SQL statements for processing message fields. For details, see **SQL Statements**.

Step 4 Set the forwarding target.

Click **Add**, select **Third-party message queue (Kafka)** from the drop-down list, set parameters by referring to **Table 2-18**, and click **OK**.

Table 2-18 Setting the forwarding target

Parameter	Description
Forwarding Target	Select **Third-party message queue (Kafka)**.
Connection Address	Specify the connection address list of the Kafka server.
Topic	Specify the topic of the forwarding target.
User Authentication Type	The default type is **PAAS**, in which the Kafka server does not authenticate users. The Kafka server also supports PLAIN and SCRAM SASL authentication. Both modes are based on username and password. SCRAM is more secure than PLAIN and includes the SCRAM-SHA-256 and SCRAM-SHA-512 algorithms.
Username	If SASL authentication is used, you need to enter the username.
Password	If SASL authentication is used, you need to enter the password.

Step 5 Enable a rule.

After the rule is configured, click **Enable Rule** to start data forwarding.

----End

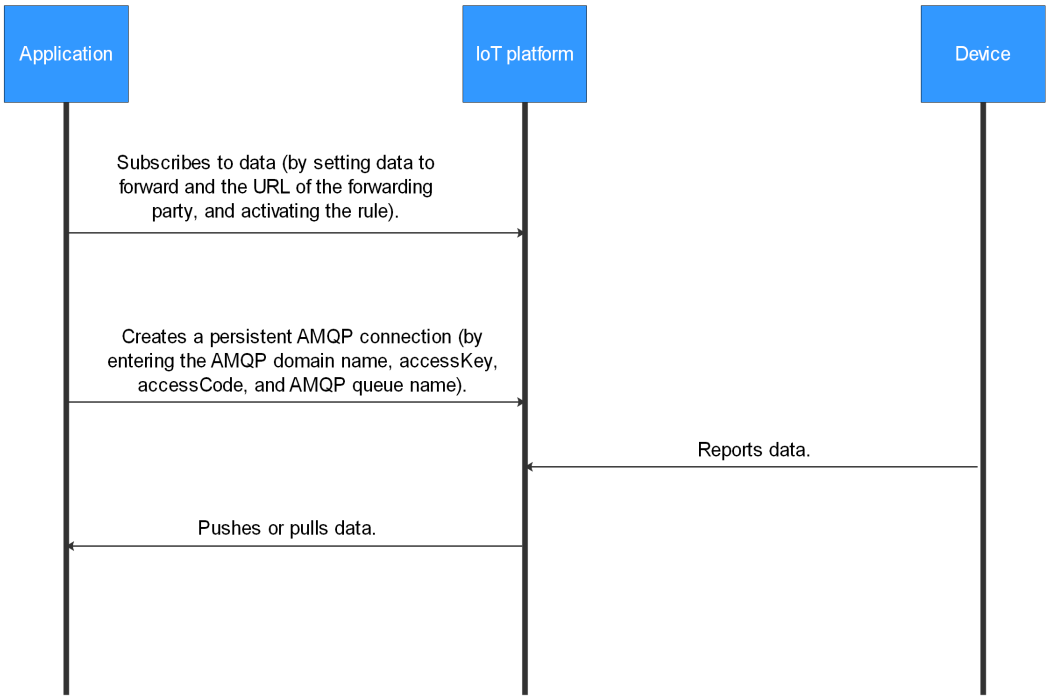
2.10.3 AMQP Subscription/Push

2.10.3.1 Overview

Subscription: AMQP is short for Advanced Message Queuing Protocol. By creating a subscription task on the IoTDA console, or calling IoTDA APIs to configure and activate rules, you can obtain changed device service details (such as device lifecycle management, device data reporting, device message status, and device status) and management details (software/firmware upgrade status and result) from IoTDA. The AMQP message channel must be specified during subscription creation.

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

Figure 2-90 AMQP subscription/push



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

2.10.3.2 AMQP Client Access

After calling APIs to configure and activate a rule, connect the AMQP client to IoTDA, and then run the AMQP client on your server to receive subscribed messages.

Protocol Version

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

IoTDA supports only the AMQP 1.0 protocol.

Connection Establishment and Authentication

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

   NOTE

   To ensure security, the AMQP client must use TLS 1.2 or a later version for encryption. Non-encrypted TCP transmission is not supported.

2. The client requests to set up a connection.

3. The client sends a request to IoTDA to establish a receiver link (a unidirectional channel for IoTDA to push data to the client).

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

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

   NOTE

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

Connection Configuration Parameters

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

- AMQP access domain name: amqps://${address}
- Connection string: amqps://${address}:5671?amqp.vhost=default&amqp.idleTimeout=8000&amqp.saslMechanisms=PLAIN

Parameter	Description
address	Address accessed by AMQP. The value can be a domain name or an IP address. Log in to the IoTDA console, choose **Overview** to obtain the AMQP access address.
amqp.vhost	Currently, AMQP uses the default host. Only the default host is supported.
amqp.saslMechanisms	Connection authentication mode. Currently, PLAIN-SASL is supported.
idle-time-out	Heartbeat interval, in milliseconds. If the heartbeat interval expires and no frame is transmitted on the connection, IoTDA closes the connection.

- Port: 5671
- Client identity authentication parameters

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

  password = "${accessCode}"

Parameter	Mandatory or Optional	Description
accessKey	Mandatory	An accessKey can be used to establish a maximum of 32 concurrent connections. When establishing a connection for the first time, preset the parameter by following the instructions provided in **Obtaining the AMQP Access Credential**.

Parameter	Mandatory or Optional	Description
timestamp	Mandatory	Indicates the current time. The value is a 13-digit timestamp, accurate to milliseconds. The server verifies the client timestamp. There is a 5-minute difference between the client timestamp and server timestamp.
accessCode	Mandatory	The value can contain a maximum of 256 characters. When you establish a connection for the first time or the accessCode is lost, obtain the access credential by following the instructions provided in **Obtaining the AMQP Access Credential**.

Obtaining the AMQP Access Credential

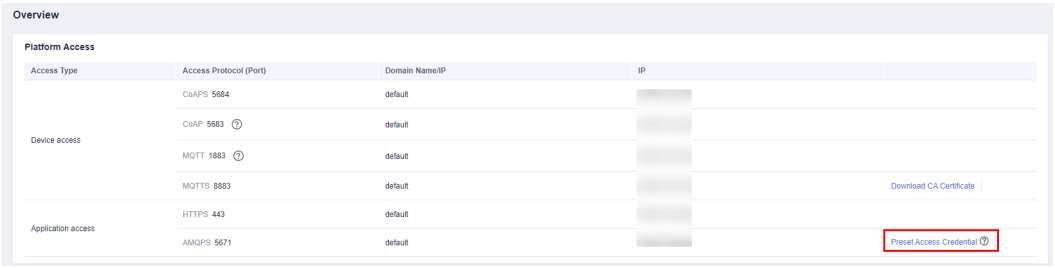
If an application uses AMQP to access IoTDA for data transfer, preset an access credential.

You can preset an access credential on the console or by calling the API **Generating an Access Credential**. The procedure for using the console to preset an access credential is as follows:

Step 1 Log in to the IoTDA console.

Step 2 In the left navigation pane, choose **Overview**. On the displayed page, click **Preset Access Credential** to preset the accessCode and accessKey.

Figure 2-91 Presetting an access credential



NOTE

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

----End

Connection Specifications

Key	Documentation
Maximum number of queues that can be connected to a connection	5
Maximum number of queues for a user	10
Maximum number of connections for a tenant	16
Maximum number of cached messages for an IoTDA instance	100
Maximum number of connections for an instance	100
Maximum number of concurrent connection requests for a single instance	50 TPS
Timeout interval from the time when a connection is created to the time when the queue subscription is successful.	30s

Receiving Push Messages

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

2.10.3.3 Java SDK Access Example

An AMQP-compliant JMS client connects to IoTDA and receives subscribed messages from IoTDA.

Requirements for the Development Environment

JDK 1.8 or later has been installed.

Obtaining the Java SDK

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

Adding a Maven Dependency

```
<!-- amqp 1.0 qpid client -->
 <dependency>
   <groupId>org.apache.qpid</groupId>
```



```
  <artifactId>qpid-jms-client</artifactId>
  <version>0.50.0</version>
</dependency>
```

Code Samples

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

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

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

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

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

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

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

        // Create a connection.
        Connection connection = cf.createConnection(userName, password);
        ((JmsConnection) connection).addConnectionListener(myJmsConnectionListener);
        // Create a session.
        // Session.CLIENT_ACKNOWLEDGE: After receiving a message, manually call message.acknowledge().
```



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

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

    }

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

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

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

        /**
         * The connection fails after the maximum number of retries is reached.
         */
```



```
    @Override
    public void onConnectionFailure(Throwable error){
        System.out.println("onConnectionFailure, " + error.getMessage());
    }

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

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

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

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

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

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

2.10.3.4 Node.js SDK Access Example

This topic describes how to use a Node.js AMQP SDK to connect to IoTDA and receive subscribed messages from IoTDA.

Development Environment

Node.js 8.0.0 or later is used.

Downloading the SDK

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

Adding Dependencies

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

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


Code Samples

Create a JavaScript file (for example, **HwIotAmqpClient.js**) on the local computer, save the following sample code to the file, and modify connection parameters by referring to **AMQP Client Access**.

- For the access domain name, see **AMQP access domain name**.
- For the method to assemble userName, see **Client identity authentication parameters**.
- For the accessCode, see **Client identity authentication parameters**.

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

// Set up a connection.
var connection = container.connect({
    //Access domain name
    'host': '{address}',
    'port': 5671,
    'transport': 'tls',
    'reconnect': true,
    'idle_time_out': 8000,
    //userName assembling method
    'username': 'accessKey=${yourAccessKey}|timestamp=' + timestamp + '|',
    //accessCode
    'password': '${yourAccessCode}',
    'saslMechannisms': 'PLAIN',
    'rejectUnauthorized': false,
    'hostname': 'default',
});

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

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

2.10.3.5 C# SDK Access Example

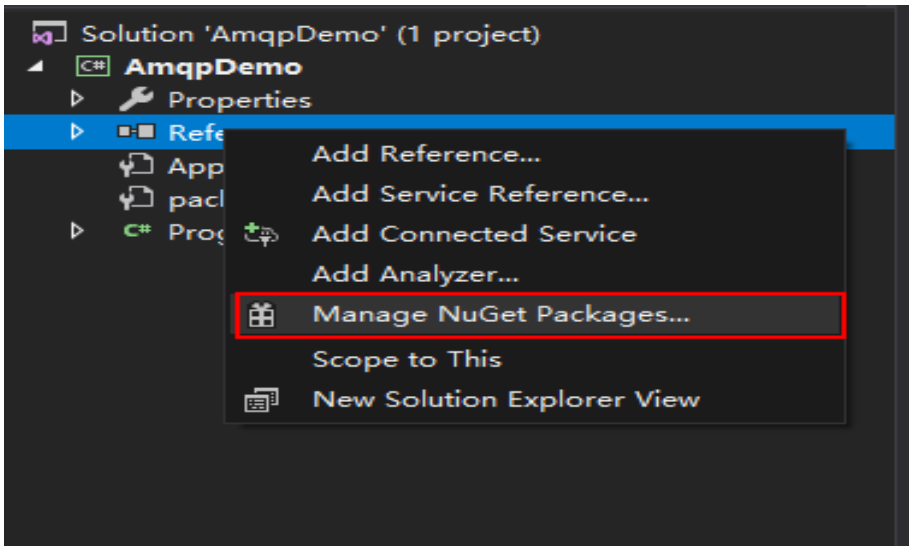
This topic describes how to connect an AMQP.Net Lite client to IoTDA and receive subscribed messages from the platform.

Development Environment Requirements

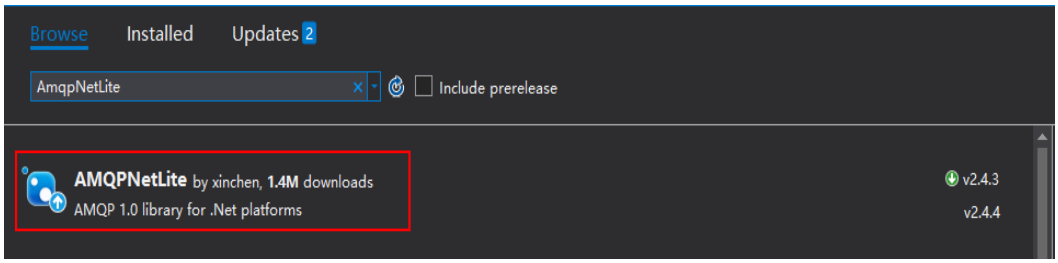
.NET Framework 4.6 or later has been installed.

Obtaining the Java SDK

1. Right-click the project directory and choose **Manage NuGet Packages**.



2. In the NuGet manager, search for **AmqpNetLite** and install the v2.4.3 version.



Sample Code

For details about the parameters in the demo, see **AMQP Client Access**.

```
using Amqp;
using Amqp.Framing;
using Amqp.Sasl;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace AmqpDemo
{
    class Program
    {
        /// <summary>
        /// Access domain name. For details, see "AMQP Client Access".
        /// </summary>
        static string Host = "${Host}";

        /// <summary>
        /// Port
        /// </summary>
        static int Port = 5671;

        /// <summary>
        /// Access key
        /// </summary>
        static string AccessKey = "${YourAccessKey}";
```



```
    /// <summary>
    /// Access code
    /// </summary>
    static string AccessCode = "${yourAccessCode}";

    /// <summary>
    /// Queue name
    /// </summary>
    static string QueueName = "${yourQueue}";

    static Connection connection;

    static Session session;

    static ReceiverLink receiverLink;

    static DateTime lastConnectTime = DateTime.Now;

    static void Main(string[] args)
    {
        try
        {
            var connection = CreateConnection();
            // Add a connection exception callback.
            connection.AddClosedCallback(ConnectionClosed);

            // Create a session.
            var session = new Session(connection);

            // Create a receiver link.
            var receiver = new ReceiverLink(session, "receiverName", QueueName);

            // Receive a message.
            ReceiveMessage(receiver);
        }
        catch (Exception e)
        {
            Console.WriteLine(e);
        }

        // Press Enter to exit the program.
        Console.ReadLine();

        ShutDown();
    }

    /// <summary>
    /// Create a connection.
    /// </summary>
    /// <returns>Connection</returns>
    static Connection CreateConnection()
    {
        lastConnectTime = DateTime.Now;
        long timestamp = new DateTimeOffset(DateTime.UtcNow).ToUnixTimeMilliseconds();
        string userName = "accessKey=" + AccessKey + "|timestamp=" + timestamp;
        Address address = new Address(Host, Port, userName, AccessCode);
        ConnectionFactory factory = new ConnectionFactory();
        factory.SASL.Profile = SaslProfile.External;
        // Trust the server and skip certificate verification.
        factory.SSL.RemoteCertificateValidationCallback = (sender, certificate, chain, sslPolicyError) =>
{ return true; };
        factory.AMQP.IdleTimeout = 8000;
        factory.AMQP.MaxFrameSize = 8 * 1024;
        factory.AMQP.HostName = "default";
        var connection = factory.CreateAsync(address).Result;
        return connection;
    }

    static void ReceiveMessage(ReceiverLink receiver)
```



```
    {
        receiver.Start(20, (link, message) =>
        {
            // Process the message in the thread pool to prevent the thread that pulls the message from
being blocked.
            ThreadPool.QueueUserWorkItem((obj) => ProcessMessage(obj), message);
            // Return an ACK message.
            link.Accept(message);
        });
    }

    static void ProcessMessage(Object obj)
    {
        if (obj is Message message)
        {
            string body = message.Body.ToString();
            Console.WriteLine("receive message, body=" + body);
        }
    }

    static void ConnectionClosed(IAmqpObject amqpObject, Error e)
    {
        // Reconnection is performed every 15 seconds.
        while (DateTime.Now.CompareTo(lastConnectTime.AddSeconds(15)) < 0)
        {
            Thread.Sleep(1000);
        }
        ShutDown();

        var connection = CreateConnection();
            // Add a connection exception callback.
        connection.AddClosedCallback(ConnectionClosed);

            // Create a session.
        var session = new Session(connection);

            // Create a receiver link.
        var receiver = new ReceiverLink(session, "receiverName", QueueName);

            // Receive a message.
        ReceiveMessage(receiver);
    }

    static void ShutDown()
    {
        if (receiverLink != null)
        {
            try
            {
                receiverLink.Close();
            }
            catch (Exception e)
            {
                Console.WriteLine("close receiverLink error, exception =" + e);
            }

        }
        if (session != null)
        {
            try
            {
                session.Close();
            }
            catch (Exception e)
            {
                Console.WriteLine("close session error, exception =" + e);
            }

        }
```



```
            if (connection != null)
            {
                try
                {
                    connection.Close();
                }
                catch (Exception e)
                {
                    Console.WriteLine("close connection error, exception =" + e);
                }

            }
        }
    }
}
```

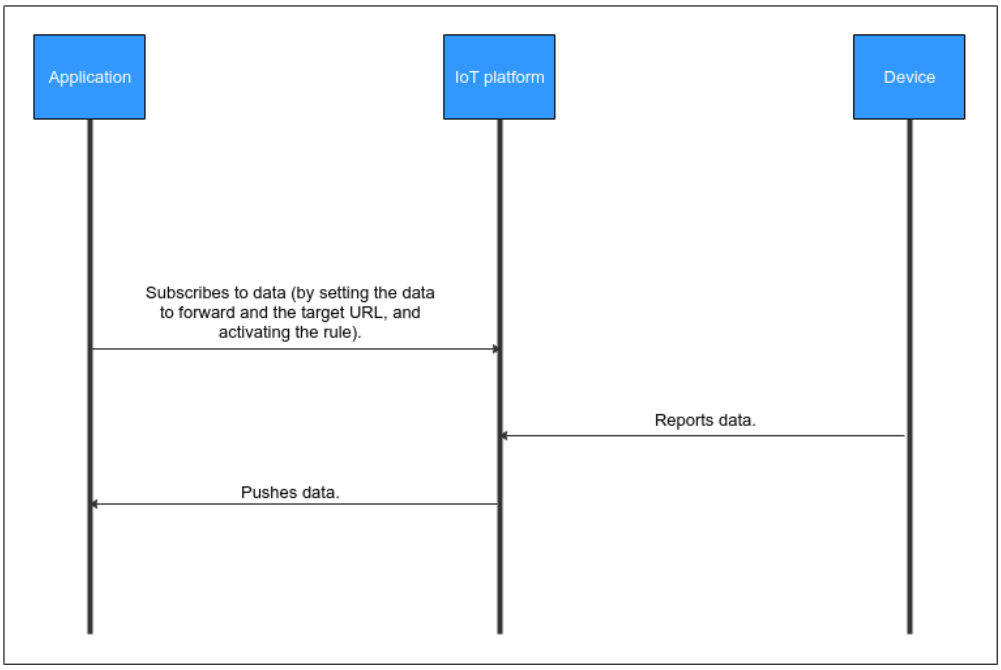
2.10.4 HTTP/HTTPS Subscription/Push

Overview

Subscription: By creating a subscription task on the IoTDA console, you can obtain changed device service details (such as device lifecycle management, device data reporting, device message status, and device status) and management details (software/firmware upgrade status and result) from IoTDA. The URL of the application, also called the callback URL, must be specified during subscription.

Push: After a subscription is successful, IoTDA pushes the corresponding change to a specified URL based on the type of data subscribed. If an application does not subscribe to a specific type of data notification, IoTDA does not push the data to the application even if the data has changed. IoTDA pushes data, in JSON format, using HTTP or HTTPS. HTTPS is an encrypted transmission protocol that requires authentication and is more secure. Therefore, HTTPS is recommended.

Figure 2-92 HTTP/HTTPS subscription/push



IoTDA verifies the application when HTTPS is used to push messages to the application. The CA certificate provided by the application must be loaded to IoTDA.

NOTE

You can during commissioning. For security reasons, you are advised to replace the commissioning certificate with a commercial certificate during commercial use.

Push mechanism: After receiving a push message from IoTDA, the application returns a 200 **OK** message. If the application does not respond within 15 seconds or returns a non-200 code (such as 500, 501, 502, 503, or 504), the message push fails. If the platform fails to push the message for 10 consecutive times, IoTDA adds the host address of the subscription URL to the blacklist and stops pushing the message to the host address. After 5 minutes, IoTDA removes the host address from the blacklist, and tries to push the message again. If the host address is still invalid, the platform will add it to the blacklist again after 10 consecutive failed pushes. If the host address is found valid, the normal push is restored.

Subscribing to Data

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

- For details on how to configure HTTP or HTTPS subscriptions on the console, see **Configuring HTTP/HTTPS Subscription** and **Loading the CA Certificate**.

Format of Pushed Data

After the data subscription is successful, IoTDA pushes the data to the application.

NOTE

In the HTTP message header, the value of **Content-Type** is **application/json**, and the character set is **UTF-8**.

Loading the CA Certificate

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

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

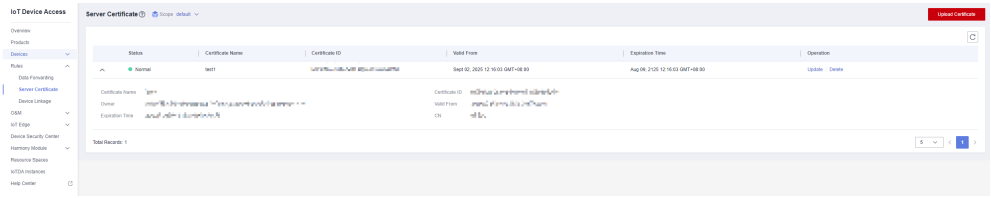
Step 1 Log in to the IoTDA console.

Step 2 In the navigation pane, choose **Rules** > **Server Certificate**. Click **Upload Certificate** in the upper right corner, configure parameters based on the following table, and click **OK**.

Parameter	Description
Certificate Name	Used to distinguish different certificates and can be customized.
CA Certificate	A CA certificate from the application can be applied for in advance. **NOTE** You can **create a commissioning certificate** during commissioning. For security reasons, you are advised to replace the commissioning certificate with a commercial certificate during commercial use.

Step 3 In the navigation pane, choose **Rules** > **Server Certificate**, locate the target certificate, and click to obtain the certificate ID, which is used as a parameter in the API for creating a rule action.

Figure 2-93 Server certificates



----End

Creating a Commissioning Certificate

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

The following table lists common certificate storage formats.

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

NOTE

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

Procedure

Step 1 Download and install **OpenSSL**.

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

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

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

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

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

Step 5 Use the private key file of the CA root certificate to generate the **ca.csr** file to be used in **6**.

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

Modify the following information based on the site requirements:

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

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

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

Modify the following information based on the site requirements:

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

Step 7 Generate the private key file for the application.

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

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

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

Modify the following information based on the site requirements:

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

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

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


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

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

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

----End

Configuring HTTP/HTTPS Subscription

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

Step 1 Log in to the IoTDA console.

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

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

Parameter	Description
Rule Name	Name of a rule to be created.
Description	Description of the rule.
Data Source	• **Device property**: A property value reported by a device in a resource space will be forwarded. • **Device message**: A message reported by a device in a resource space will be forwarded. • **Device message status**: The status of device messages exchanged between the device and platform will be forwarded. • **Device status**: The status change of a directly connected device in a resource space will be forwarded. • **Device event**: Only events you defined in the product will be forwarded. • **Batch task**: The batch task status will be forwarded. • **Product**: Product information, such as product addition, deletion, and update, will be forwarded. • **Device**: Device information, such as device addition, deletion, and update, will be forwarded. • **Device alarm**: Device alarm information, such as device alarm generation and clearance, will be forwarded. • **Asynchronous command status of the device**: The command status change of the device will be forwarded.

Parameter	Description
Trigger	After the data source is selected, the platform automatically matches the trigger event.
Resource Space	You can select a single resource space or all resource spaces.
SQL Filter Statements	You can edit the SQL statements for processing message data and set the data filtering statements. Click **Edit SQL** to edit the SQL statements for processing message fields. For details, see **SQL Statements**.

Step 4 Set the forwarding target.

Click **Add**, select **Third-party application (HTTP push)**, set parameters by referring to **Table 2-19**, and click **OK**.

Table 2-19 Setting the forwarding target

Parameter	Description
Forwarding Target	Select **Third-party application (HTTP push)**.
Push URL	URL used by IoTDA to push messages to an application. For example, if the push URL is **https://10.10.10.10:8443/example/**, the domain name/IP address and port number for **Loading the CA Certificate** are **10.10.10.10:8443**. • If the push URL uses HTTP, the CA certificate is not required. • If the push URL uses HTTPS, upload the CA certificate. For details about how to upload a certificate, see **Loading the CA Certificate**.
Encryption	Whether to enable the certificate to encrypt data. This parameter is enabled by default. **NOTICE** By default, HTTPS authentication is used. IoTDA also supports HTTP non-authentication mode, which may cause data leakage because it does not support transport layer encryption. IoTDA does not make any guarantee or assume any liability for your device data.
Encryption Certificate	Select the server certificate uploaded in **Loading the CA Certificate**.

Step 5 Enable a rule.

After the rule is configured, click **Enable Rule** to start data forwarding.

----End

2.11 IoTEdge

2.11.1 Node Management

2.11.1.1 Registering an Edge Node

IoTEdge provides basic edge management functions required for the IoT industry.

Prerequisites

Before the registration, check the host software and hardware configuration, such as the disk, memory, and Docker version. In addition, user **root** is required to run installation commands.

Careful checks prevent installation failures or service unavailability after the installation.

Registering a Node

Step 1 Log in to the IoTDA console.

Step 2 Choose **IoTEdge** > **Nodes** in the left navigation pane. On the page displayed, click **Register Node** in the upper right corner.

Step 3 Configure node specifications by referring to **Table 2-20**. Agree to the Cloud Service Level Agreement and click **Next**.

Table 2-20 Configuring node parameters

Parameter	Description
Node Type	Professional edition
Node Name	Custom the name of an edge node to register. Enter 1 to 64 characters. Only letters, numbers, hyphens (-), and underscores (_) are allowed.
Gateway Type	other
Node ID	(Optional) Set this parameter as required. The value can contain 1 to 64 characters, including letters, digits, hyphens (-), and underscores (_). **NOTE** Customize an edge node ID. If this parameter is not specified, the edge node ID will be generated by default.
Verification Code	(Optional) Enter a custom verification code, which is used as a credential for the communication with the cloud when the edge software installation command is executed.

Parameter	Description
Bind IoTDA	Select the resource space to which the node belongs.
Resource Space	Select the default resource space or create a resource space, for example, **edge**.
Authentication Mode	**Secret**: generated by default. **X.509 certificate**: For details, see **MQTT X.509 Certificate Access**.
Fingerprint	(Optional) Enter the string generated by the certificate file. Enter a 40-digit or 64-digit hexadecimal string.
Local Path	If fingerprint authentication is used, enter the storage paths of the certificate and key files on the device. The value can contain only letters, digits, underscores (_), and slashes (/), and must start with a slash (/) and end with .pem or .crt.

Step 4 Configure node data by referring to **Table 2-21** and click **Create Now**.

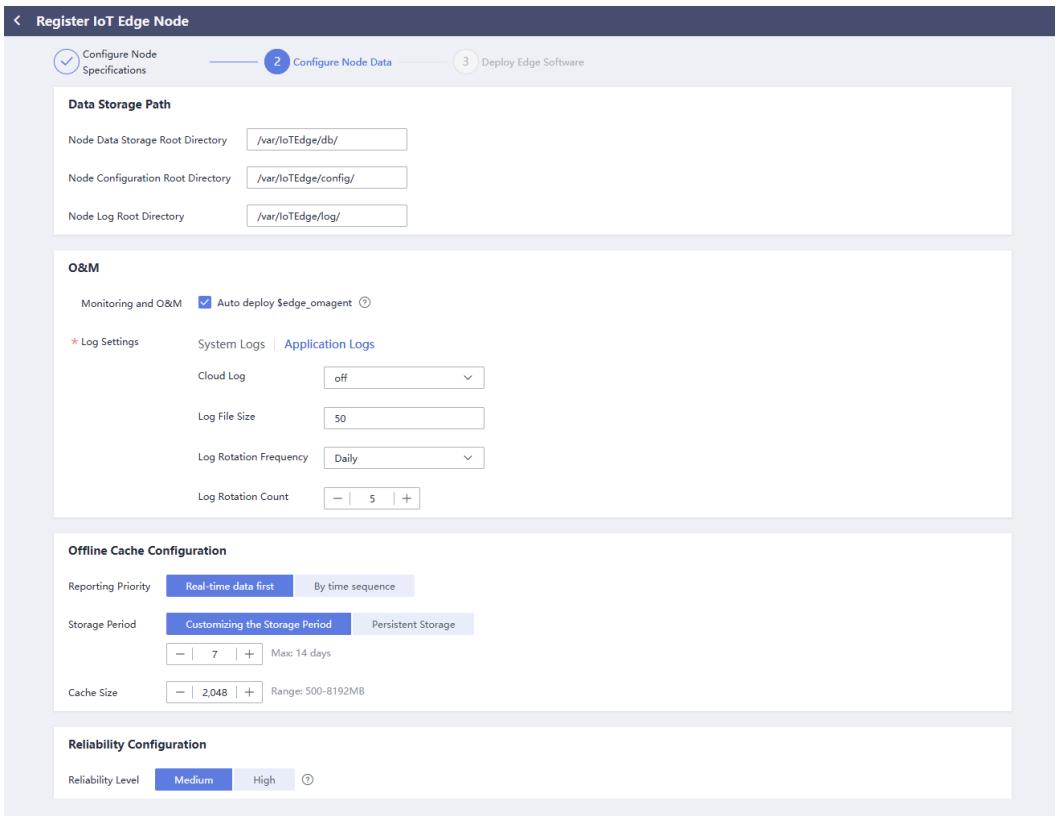
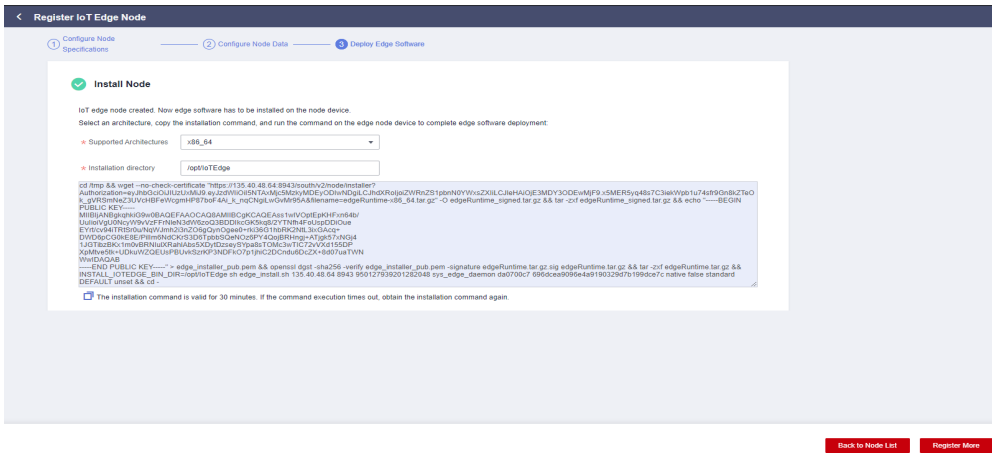


Table 2-21 Configuring node data

Parameter	Description
Data Storage Path	Retain the default value (the storage path can be changed).

Parameter	Description
Monitoring and O&M	The monitoring O&M tool is selected by default.
$edge_omagent	It performs remote monitoring and O&M on edge nodes. It reports data (such as logs and CPU indicators) and provides remote SSH connection and file upload/download.
Log Settings	**System Logs**: logs generated by system applications deployed on edge nodes. **Application Logs**: logs generated by custom applications deployed on edge nodes. • **Cloud Log**: **On** indicates that logs are uploaded to the cloud, and **Off** indicates that logs are not uploaded. • **Log File Size**: maximum size of a log file, in MB. The default value is **50**. The value ranges from 10 to 1000. If a log file reaches the specified size, the system dumps the log file into the specified directory. • **Log Rotation Frequency**: frequency at which logs are dumped into the specified directory. The value can be **Daily**, **Weekly**, **Monthly**, or **Yearly**. If a log file reaches the frequency, the system dumps the log file into the specified directory. Log files will be dumped when either **Log File Size** or **Log Rotation Frequency** is reached. • **Log Rotation Count**: maximum number of logs being retained. The default value is **5**. The value ranges from 1 to 10. Once this number is reached, when new log files are dumped in, old log files will be deleted on a FIFO basis.
Offline Cache Configuration	You can cache data reported by offline nodes and define the cache period (which can be set to permanent storage), cache size, and reporting priority.
Reliability Configuration	Options are **Medium** and **High**. **Medium** indicates that data is directly discarded after the threshold for edge traffic control is reached, which is used for common scenarios. **High** indicates that data is not discarded after the threshold is reached, but the application sending speed is controlled within the threshold and the latency increases. (Note: **High** requires a custom synchronous sending application.)

Step 5 In the dialog box displayed, select the supported architecture and enter the installation directory, which is used to store logs. You can go back to the node list or continue to create a node.



NOTE

1. Check the **ip_forward** value in the **/proc** file system.
   ```
   cat /proc/sys/net/ipv4/ip_forward
   ```
   If the value is **0**, add the IP forwarding rule configuration.
2. Modify the **/etc/sysctl.conf** configuration file and set **net.ipv4.ip_forward** to **1**.
   If the configuration file does not contain the **net.ipv4.ip_forward** parameter, add **net.ipv4.ip_forward = 1**.
   ```
   vim /etc/sysctl.conf
   ```
   Press **i** to enter the editing mode. After the modification is complete, press **Esc** to exit the editing mode. Then, press **Shift+;**, enter **wq**, and press **Enter** to save the modification and exit.
3. Reload the file.
   ```
   sysctl -p
   ```
4. Check whether ports 7882, 7883, and 8943 are enabled.

----End

2.11.1.2 Installing an Edge Node

Prerequisites

Edge software has the following requirements on hardware, operating environment, and network:

Hardware requirements

Hardware	Requirement
OS	• x86_64 architecture EulerOS 2.3 EulerOS 2.9 Ubuntu 16.04 Ubuntu 18.04 CentOS 7.6 • Arm64 EulerOS 2.8 EulerOS 2.9 Ubuntu 18.04
CPU	≥ 1 vCPU
RAM	≥ 256 MB (100 TPS for 1U1G; 1,000 TPS for 8U8G)
Disk	≥ 2 GB
GPU (optional)	The GPU models on the same edge node must be the same.

Operating environment requirements

Dependency Item	Specifications
Docker	The Docker version must be later than 17.06. Docker 18.06.3 is recommended. (However, do not use Docker 18.09.0 because it has a serious bug. For details, see **https://github.com/docker/for-linux/issues/543**. If this version has been used, upgrade it as soon as possible. This issue has been resolved in Docker 18.09.0.60 used by Atlas 500 AI edge stations.) For details about how to install Docker, see **https://docs.docker.com/install/overview/**. You can install open-source Docker Engine - Community (Docker CE) or paid Docker Engine - Enterprise (Docker EE). For more details about Docker EE, see the official Docker documentation at **https://docs.docker.com/ee/supported-platforms/**. **NOTE** After Docker is installed, configure the Docker process to start upon host startup. This configuration prevents system exceptions that occur when the Docker process does not start together with the host.
wget	The version must be 1.10 or later.
openssl	The version must be 1.0.2 or later.
Port	Edge nodes need to use the following ports. Ensure that these ports function properly. • 7882: provides southbound MQTT device access. • 7883: provides southbound MQTTS device access. • 8943: HTTP port
NTP (optional)	The NTP server must be reliable, and the time difference must be less than or equal to 5 seconds.

Network requirements

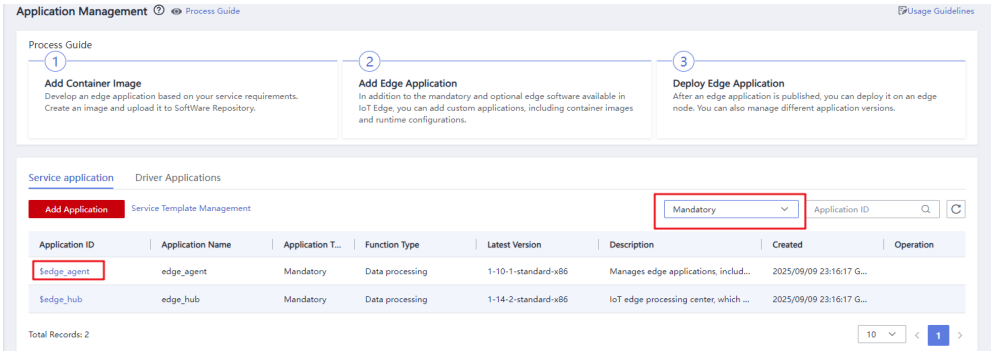
The VM where the edge node is installed and IoTDA are in the same VPC.

Step 1 Configure an EIP address for the VM where the edge node is installed.

Step 2 Obtain the IoTDA private repository address.

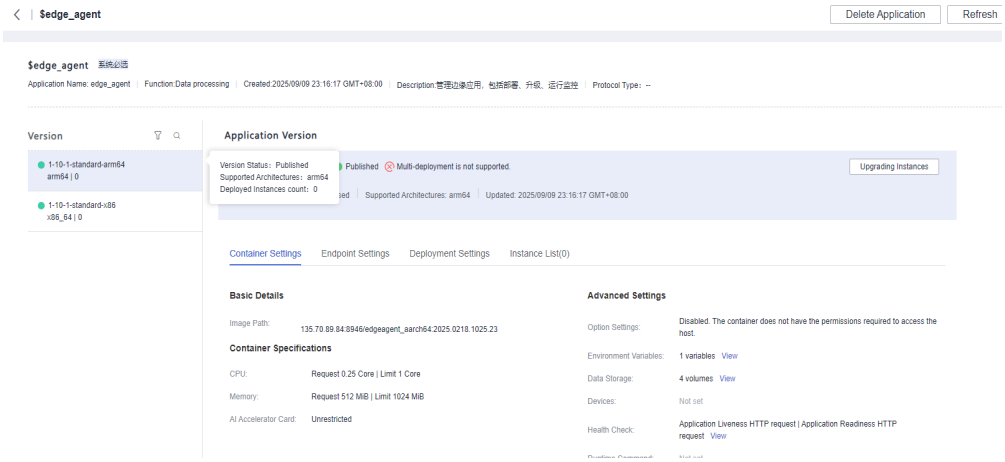
1. Log in to the IoTDA console. In the navigation pane, choose **IoTEdge** > **Applications**. Select **Mandatory** from the drop-down list for filtering and click the **$edge_agent** application.

Figure 2-94 Application management



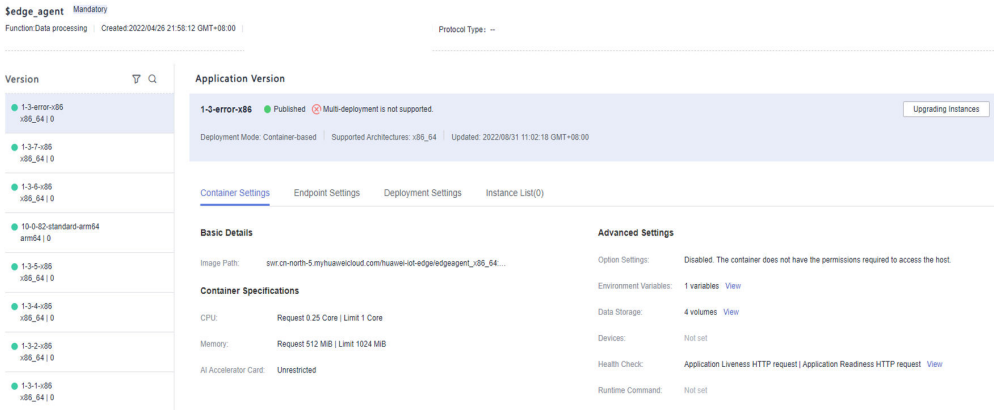
2. Select the target version and view the version information on the right.

Figure 2-95 Service version



3. The address in the red box is *{Private repository address}*.

Figure 2-96 Checking the private repository address



Step 3 Log in to the VM where the edge node is to be installed, modify the Docker configuration file **/etc/docker/daemon.json**, find the **insecure-registries** configuration, and add *{Private repository address}* obtained in the previous step.

For example:

```
vim /etc/docker/daemon.json
{
```



```
"insecure-registries":["{Private repository address}"]
}
```

If the **/etc/docker/daemon.json** file does not exist, run the following command to create the file:

```
touch /etc/docker/daemon.json
```

Copy the following content to the file and save the file:

```
{
   "insecure-registries": ["{Private repository address}"],
   "default-ulimits": {
      "nofile": {
         "Name": "nofile",
         "Hard": 1000000,
         "Soft": 1000000
      }
   },
   "log-driver": "json-file",
   "log-opts": {
      "max-size": "10m",
      "max-file": "5"
   }
}
```

Step 4 Run the following commands to restart Docker:

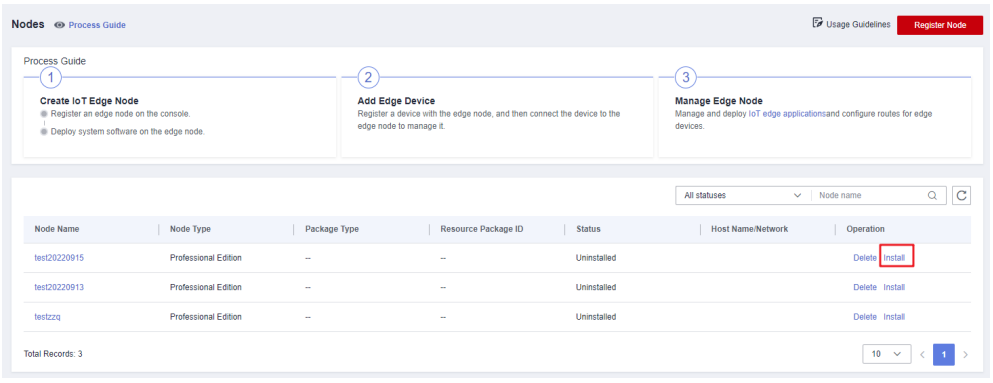
```
systemctl daemon-reload
systemctl restart docker.service
```

----End

Installing a Node

Step 1 Choose **IoTEdge** > **Nodes**. In the edge node list, locate the node to be installed and click **Install** on the right.

Figure 2-97 Installing a node



Step 2 Select the supported architecture, click to copy the installation command, use the SSH tool to log in to the background system of the edge node, and run the installation command.

Figure 2-98 Prompt

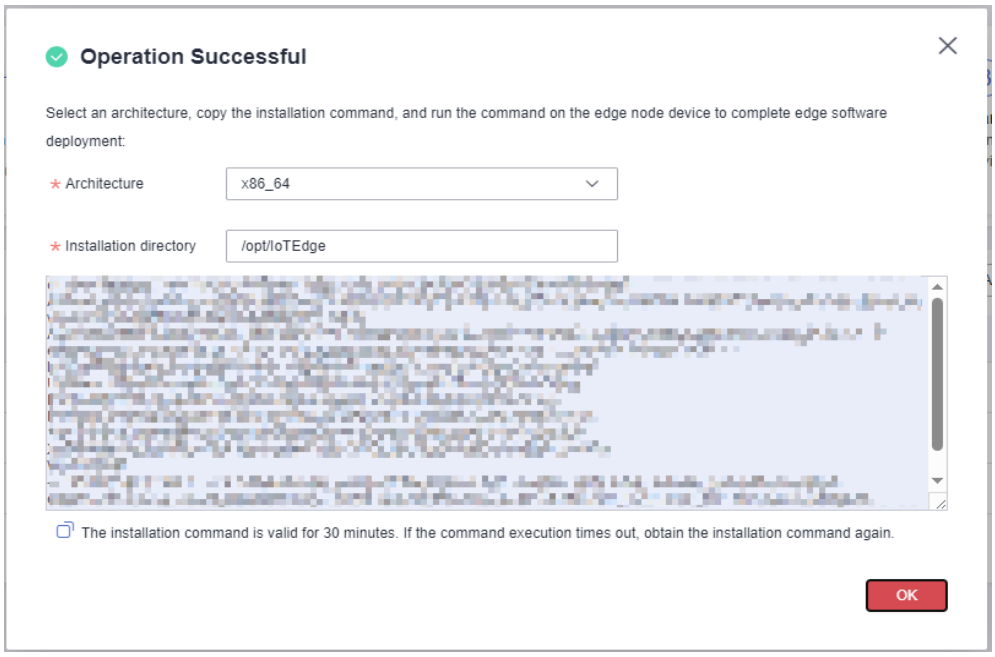
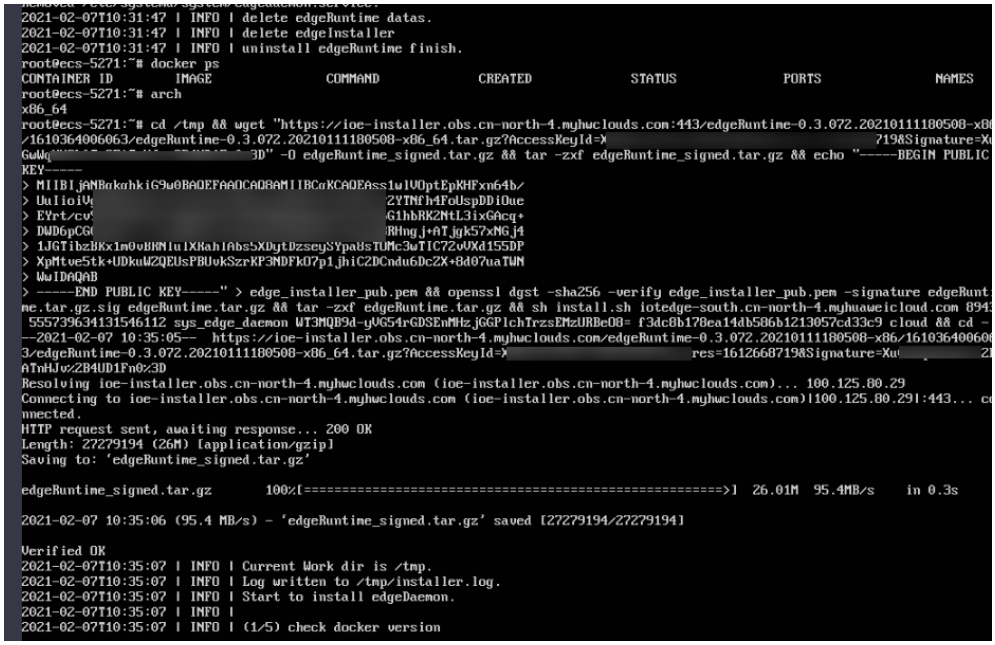


Figure 2-99 Running the installation command



Step 3 Click **OK** and wait until the status of the edge node changes to **Online**. This status indicates that the node is installed.

Step 4 Click the node name to view its details. For details, see **Overview**.

----End

2.11.1.3 Managing an Edge Node

2.11.1.3.1 Overview

Choose **IoTEdge** > **Nodes** and click the edge node name to go to the node details page.

Figure 2-100 Node details

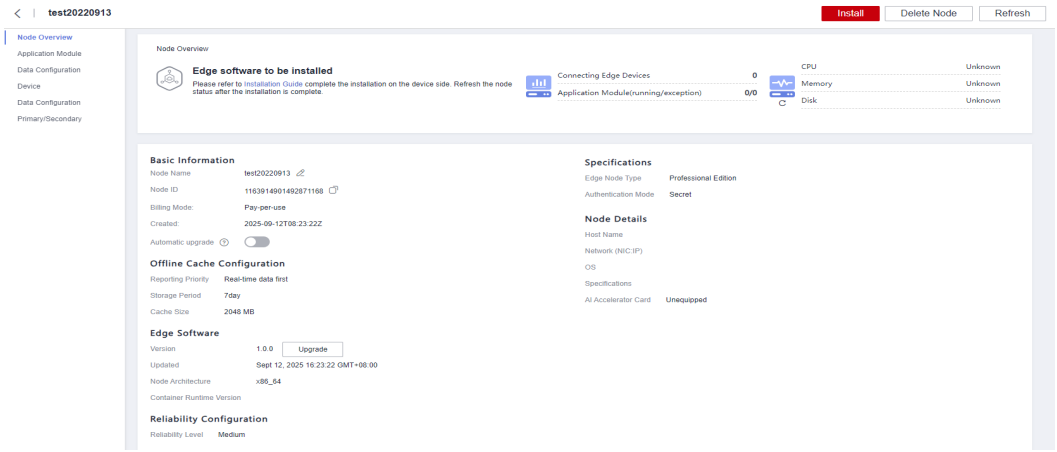


Table 2-22 Node statuses

Status	Description
Uninstalled	The edge node is not installed.
Installed	The node is installed, but the edge service is not running.
Online	The edge service is running properly.
Offline	The edge node is unavailable, and the status of modules and devices under the edge node is not updated.
Deleting	The edge node is being deleted.

Table 2-23 Overview

Parameter	Description
Node Overview	The following information is displayed: • Number of connected edge devices • Number of application modules (running/abnormal) • CPU, memory, and hard disk information
Basic Information	The following information is displayed: • Basic information • Node ID • Billing mode • Creation time

Parameter	Description
Offline Cache Configuration	The following information is displayed: • Reporting priority • Storage period • Cache size
Specifications	The following information is displayed: • Type of the edge node • Authentication mode
Edge Software	The following information is displayed: • Version • Update time • Node architecture • Container runtime version
Reliability Configuration	The reliability level is displayed.
Node Details	The following information is displayed: • **Host Name**: Linux host name of the edge node • **Network (NIC:IP)**: NIC and IP address list of the edge node • **OS**: Linux OS name of the edge node • **Specifications**: the number of CPU cores and memory size of the edge node • **AI Accelerator Card**

Monitoring Resources

Prerequisites

To use the resource monitoring function, deploy the $edge_omagent application.

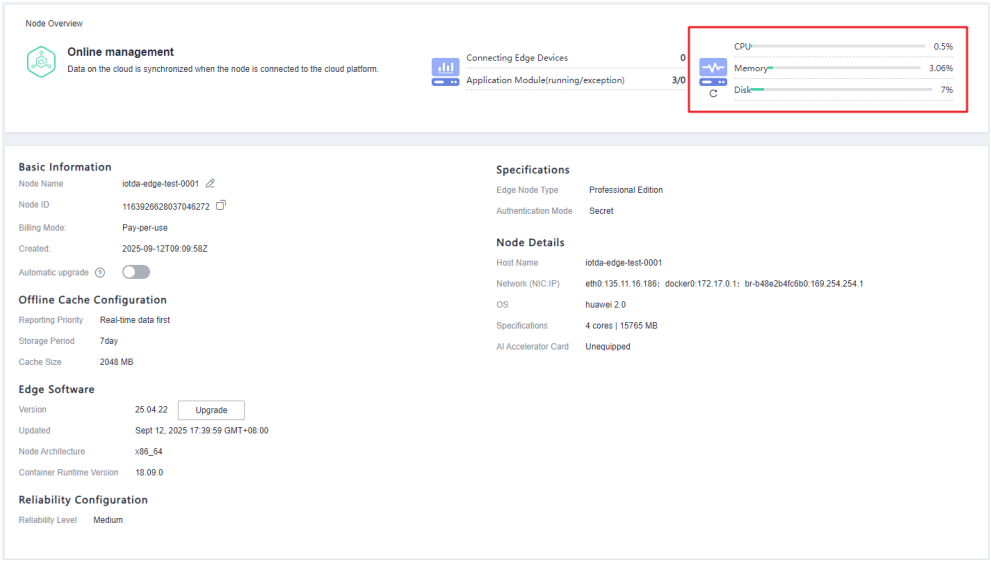
For details on how to deploy edge applications, see **Deploying an Application**.

Procedure

Step 1 Choose **IoTEdge** > **Nodes** and click the edge node name to go to the node details page.

Step 2 In the upper right corner of the page, the CPU, memory, and disk information of the node is displayed.

Figure 2-101 Basic Information of a node



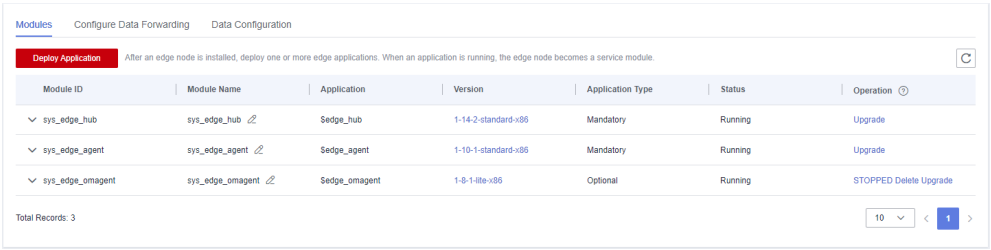
----End

2.11.1.3.2 Modules

Choose **IoTEdge** > **Nodes**, click the edge node name to go to the node details page, and choose **Application Module** on the left to deploy an edge application and configure data forwarding.

- For details on how to deploy edge applications, see **Deploying an Application**.
- For details on how to configure data forwarding, see **Configuring Data Forwarding**.

Figure 2-102 Deploying an edge application



Configuring Data Forwarding

You can configure the source and destination of data forwarding as required so that messages can be forwarded to the corresponding endpoint based on the specified path, improving data security.

By default, a data forwarding rule forwards data from a device to IoTDA. Pay attention to the following points when configuring data forwarding:

CAUTION

- Only applications that are configured with input and output endpoints can use data forwarding.
- After a module is deleted or upgraded, data forwarding may fail. Adjust the rules in a timely manner.

The procedure is as follows:

Step 1 Log in to the IoTDA console.

Step 2 Choose **IoTEdge** > **Nodes** in the left navigation pane.

Step 3 Click the target edge node name to access its details page.

Step 4 Choose **Application Module** and click **Configure Data Forwarding**.

Step 5 Click **Configure Rule** and then **Add Rule** to add a data cleansing rule based on **Table 2-24**.

Figure 2-103 Configuring data forwarding

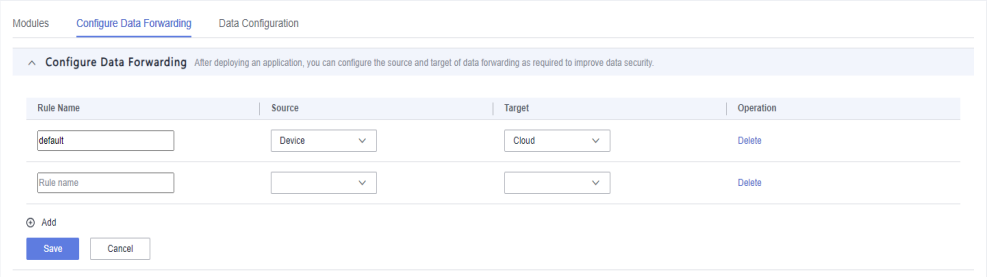


Table 2-24 Parameters for adding a data forwarding rule

Parameter	Description
Rule Name	Enter the name of a data forwarding rule.
Source	Output endpoint of the module that sends messages and the application to which the module belongs.
Target	Input endpoint of the module that receives messages and the application to which the module belongs.
Delete	You can delete a data forwarding rule that is no longer required. After a rule is deleted, the system will not forward messages destined for the specified resource of the source endpoint.

Step 6 Click **Save**.

----End

2.11.1.3.3 OT Data Collection Configuration

Prerequisites

To configure OT data collection, **create a product** on IoTDA. A product is a collection of devices with the same capabilities or features.

After creating a product, add the corresponding edge device to the edge node to receive the OT device data collected by the edge node.

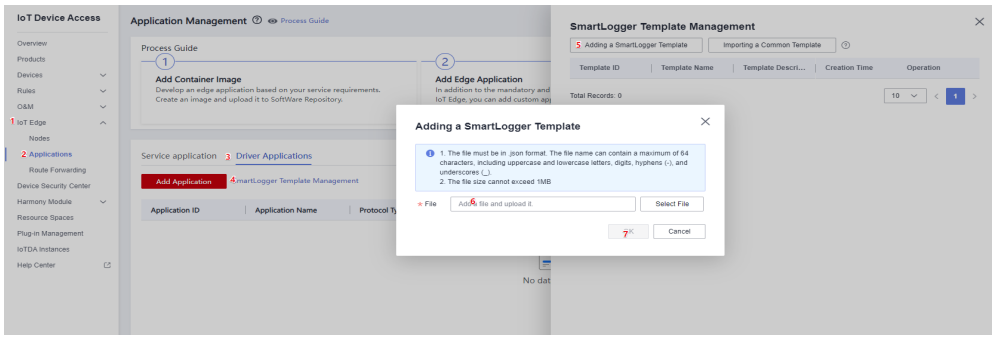
Collecting OT Data using IoTEdge

IoTEdge provides simple configuration on the console for OT data collection (from DCS, PLC, and OPC UA).

The following uses OPC UA as an example to demonstrate how to configure data collection. Before configuring data collection, deploy an application. For details, see **Deploying an Application**.

Step 1 Choose **IoTEdge** > **Applications**, click the **Driver Applications** tab, and click **Template Management**. On the displayed page, import a preset general template or upload a data collection template.

Figure 2-104 Adding a data collection template



The template file is in JSON format. The example and field description are as follows:

Example

```
{
   "tpl_id": "sys_general_opcua",
   "name": "OPC UA general data collection template",
   "description": "OPC UA general data collection template",
   "datasource_meta": {
      "config_tabs": [{
         "key": "connection_info",
         "name": "Connection information",
         "description": "Connection information",
         "config_items": [{
            "key": "endpoint",
            "name": "Service endpoint",
            "description": "Complete service URL",
            "data_type": "string",
            "required": true,
            "crypted": false,
            "max_length": 128,
            "example": "opc.tcp://127.0.0.1:53530/OPCUA"
```



```
        }, {
            "key": "username",
            "name": "Username",
            "description": "OPC UA server authentication username",
            "data_type": "string",
            "required": false,
            "crypted": false,
            "max_length": 128,
            "example": "admin"
        }, {
            "key": "password",
            "name": "Password",
            "description": "OPC UA server authentication password",
            "data_type": "string",
            "required": false,
            "crypted": true,
            "max_length": 512,
            "example": "******"
        }]
    }, {
        "key": "collection_paras",
        "name": "Additional parameters for connection",
        "description": "Additional parameters for connection",
        "config_items": [{
            "key": "default_cycle",
            "name": "Collection period",
            "description": "Collection period",
            "data_type": "int",
            "required": true,
            "crypted": false,
            "max_length": 65535,
            "example": 10000
        }]

    }],
    "default_values": {
        "drivername": "OPCUA"
    }
},
"point_meta": {
    "config_items": [{
        "key": "address",
        "name": "Point address",
        "description": "Collection point address",
        "data_type": "string",
        "required": true,
        "crypted": false,
        "max_length": 256,
        "example": "ns=2;s=Root/Motor/Voltage"
    }, {
        "key": "data_type",
        "name": "Data type of the point value",
        "description": "Data type of the point address",
        "data_type": "string",
        "required": true,
        "crypted": false,
        "max_length": 128,
        "example": "int,decimal"
    }, {
        "key": "cycle",
        "name": "Point collection period",
        "description": "Point collection period",
        "data_type": "int",
        "required": true,
        "crypted": false,
        "max_length": 65535,
        "example": 10000
    }]
```



```
  }
}
```

Table 2-25 OT data collection template

Key	Type	Description
tpl_id	String	Data collection template ID, which is unique for a tenant. Pattern: '^[a-zA-Z0-9_-]*$' Value length: 1–64 characters
name	String	Data source template name (English). Value length: 1–64 characters
description	String	Description of the data source template. The value can contain 0 to 128 characters.
datasource_meta	Object	Data source configuration metadata.
point_meta	Object	Point collection configuration metadata.

Table 2-26 datasource_meta

Key	Type	Description
config_tabs	Array	Data source configuration table list.
default_values	Object	Default data source value, which is usually used to describe the default driver protocol.

Table 2-27 config_tabs

Key	Type	Description
key	String	Key of the data source configuration table. Options include **connection_info** and **collection_paras**.
name	String	Name of the data source configuration table. The value can contain 1 to 64 characters.
description	String	Description. The value can contain 0 to 255 characters.
config_items	Array	Configuration item list.

Table 2-28 config_items

Key	Type	Description
key	String	Configuration item key. The value can contain 1 to 32 characters.
name	String	Default item name. The value can contain 1 to 64 characters.
description	String	Configuration item description. The value can contain 0 to 128 characters.
data_type	String	Configuration item type. The value can be **short**, **ushort**, **int**, **int64**, **uint**, **long**, **ulong**, **float**, **double**, **bool**, **string**, **object**, and **decimal**.
required	Boolean	Whether a configuration item is mandatory.
crypted	Boolean	Whether a configuration item is encrypted.

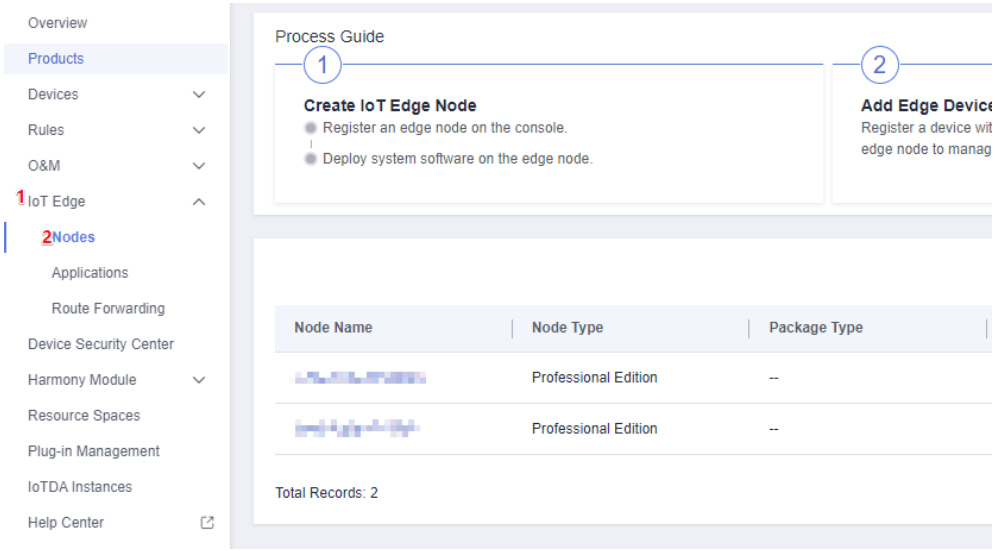
Key	Type	Description
max_length	Integer	Maximum length of an input string. This parameter is valid only when **data_type** is set to **string**.
example	String	Example of a configuration item. The value is displayed in gray in the text box on the GUI. The value can contain 0 to 256 characters.

Table 2-29 default_values

Key	Type	Description
drivername	String	Default driver protocol, for example, OPC UA.

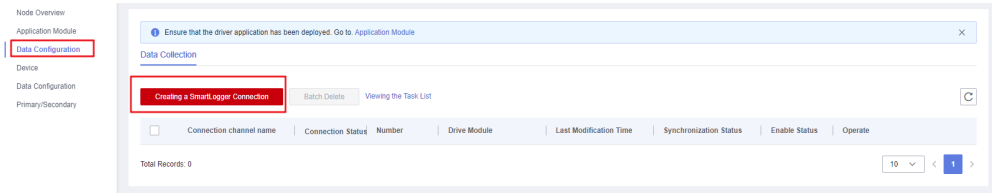
Step 2 In the navigation pane on the left, choose **IoTEdge** > **Nodes**.

Figure 2-105 Node management



In the navigation pane on the left, choose **Data Configuration** and create a data collection connection.

Figure 2-106 Data collection configuration



Specify parameters of the data collection connection and click **OK**. The configuration is only saved on the platform and is not delivered to the edge.

- Channel ID: opcua
- Channel Name: opcua (custom)
- Driver Module: user_opcua
- Service Endpoint: opc.tcp://121.36.62.255:53530/OPCUA/SimulationServer
- Username: same as that of the OPC UA simulator
- Password: same as that of the OPC UA simulator
- Collection Period: 1,000 ms

NOTE

Set service endpoint to the connection address provided by the OPC UA server. Common OPC UA servers include SCADA and Kepware.

Step 3 In the navigation pane on the left, choose **Data Configuration**, and select the target data collection connection.

Click the button for adding a point, enter configuration information by referring to **Table 2-30**, and click **OK** to add a point mapping.

Figure 2-107 Adding a point mapping

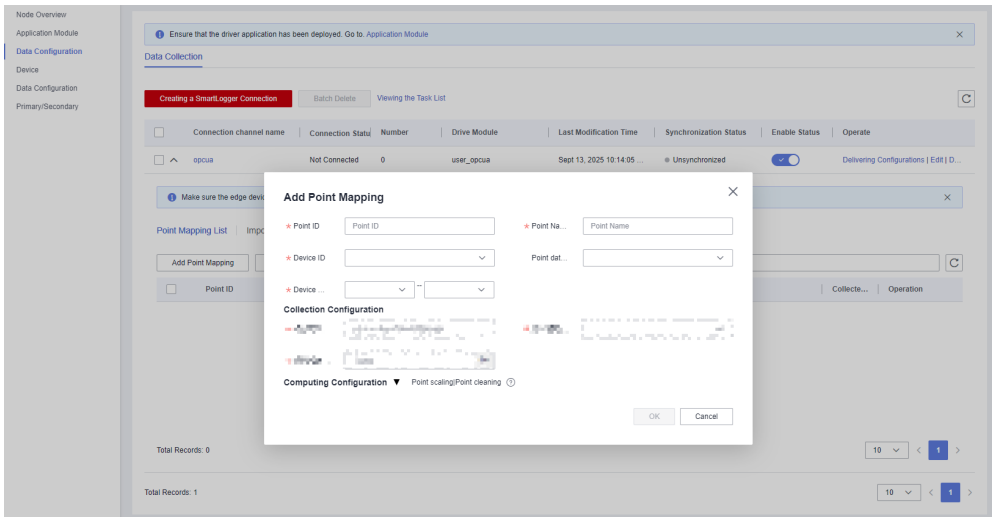


Table 2-30 Adding a point mapping

Basic information	
Point ID	Unique ID of a point.

Basic information		
Point Name	Set this parameter according to the rules.	
Device ID	ID of the device added on the edge device tab page.	
Data Type	Currently, the following data types are supported: int uint (unsigned integer) int64 (64-bit integer) short (short integer) ushort (unsigned short integer) long (long integer) ulong (unsigned long integer) bool (Boolean) float (single-precision floating point) double (double-precision floating point) decimal string object	Value range: –2147483648 to 2147483647 0 to 4294967295 –2147483648 to 2147483647 –32768 to 32767 0 to 65535 –2147483648 to 2147483647 0 to 4294967295 0/1 –3.4 x 10^38 to 3.4 x 10^38 –1.7 x 10^-308 to 1.7 x 10^308 –1.7 x 10^-308 to 1.7 x 10^308
Device Property	A device property is in the format of service_id/property_name in the product model.	
Collection configuration		
Point Address	Enter the actual point addresses of the OT device and system: **ns=3;i=1001** and **ns=3;i=1002**.	

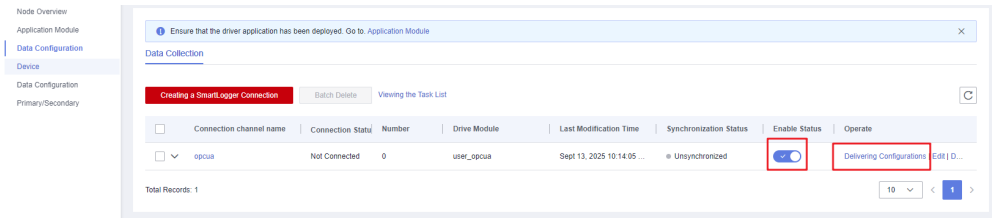
Basic information		
Data Type	Currently, the following data types are supported: int uint (unsigned integer) int64 (64-bit integer) short (short integer) ushort (unsigned short integer) long (long integer) ulong (unsigned long integer) bool (Boolean) float (single-precision floating point) double (double-precision floating point) decimal string object	Value range: –2147483648 to 2147483647 0 to 4294967295 –2147483648 to 2147483647 –32768 to 32767 0 to 65535 –2147483648 to 2147483647 0 to 4294967295 0/1 –3.4 x 10^38 to 3.4 x 10^38 –1.7 x 10^-308 to 1.7 x 10^308 –1.7 x 10^-308 to 1.7 x 10^308
Point Collection Period	1,000 ms	

CAUTION

Do not configure the same property for the same device for different points.

Step 4 Deliver the point configuration to the edge.

Figure 2-108 Delivering configuration



----End

NOTE

1. After modifying point information, you need to deliver the configuration again for it to take effect.
2. The module status is determined by the connection status. After the data collection connection is enabled, the module is displayed as running.

2.11.1.3.4 IT Data Collection Configuration

Choose **IoTEdge** > **Nodes**, click the edge node name to go to the node details page, and choose **Data Configuration** on the left to manage configuration items of third-party applications.

Collecting IT Data using IoTEdge

IoTEdge supports both IoT device data collection and subsystem data collection, implementing efficient collection and precise delivery of diverse subsystem data.

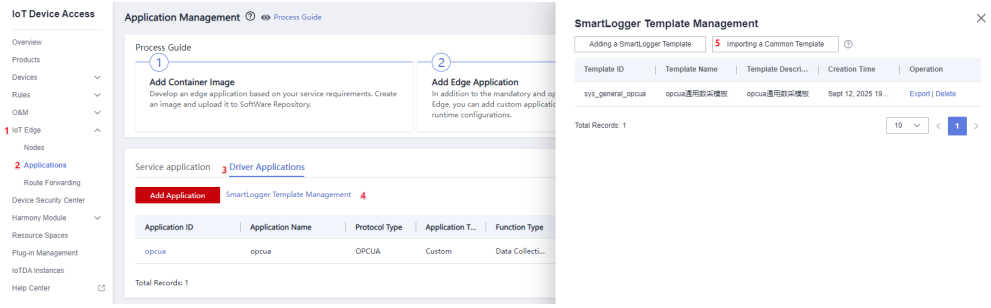
After the Enterprise Resource Planning (ERP) and Manufacturing Execution System (MES) subsystems are connected using IoTEdge, you can configure the preset integration template (ERP-MES) to efficiently collect data from the ERP subsystem. Processed data can be synchronized to the MES subsystem for subsequent management.

CAUTION

- The ERP and MES edge data collection template connects only to a simulated IT system. You need to configure the templates on the onsite IT system.
- A driver application is bound to a data collection template. You need to add a driver application developed for the template based on the IT system type.

Step 1 Choose **IoTEdge** > **Applications**, and click the **Driver Applications** tab. Access the data collection template management page, and import a common data collection template. Customize a data collection template based on the ERP and MES data collection template, and upload the custom template.

Figure 2-109 Adding a data collection template



Example of an IT data collection template:

```
{
  "tpl_id": "sys_general_erp_mes",
  "name": "General data collection template for ERP and MES integration",
```



```
"description": "General data collection template for ERP and MES integration",
"datasource_meta":
{
    "config_tabs": [
    {
        "key": "connection_info",
        "name": "Connection information",
        "config_items": [
        {
            "key": "erp_url",
            "name": "erp_url",
            "description": "Address",
            "data_type": "string",
            "required": true,
            "crypted": false,
            "max_length": 200,
            "example": ""
        },
        {
            "key": "erp_username",
            "name": "erp_username",
            "description": "Username",
            "data_type": "string",
            "required": false,
            "crypted": false,
            "max_length": 200,
            "example": ""
        },
        {
            "key": "erp_password",
            "name": "erp_password",
            "description": "Password",
            "data_type": "string",
            "required": false,
            "crypted": true,
            "max_length": 200,
            "example": ""
        },
        {
            "key": "mes_url",
            "name": "mes_url",
            "description": "Address",
            "data_type": "string",
            "required": true,
            "crypted": false,
            "max_length": 200,
            "example": ""
        },
        {
            "key": "mes_username",
            "name": "mes_username",
            "description": "Username",
            "data_type": "string",
            "required": false,
            "crypted": false,
            "max_length": 200,
            "example": ""
        },
        {
            "key": "mes_password",
            "name": "mes_password",
            "description": "Password",
            "data_type": "string",
            "required": false,
            "crypted": true,
            "max_length": 200,
            "example": ""
        }]
    },
```

```
    {
        "key": "collection_paras",
        "name": "Additional parameters for connection",
        "description": "Additional parameters for connection",
        "config_items": [
        {
            "key": "default_cycle",
            "name": "Collection task",
            "description": "Collection task",
            "data_type": "string",
            "required": true,
            "crypted": false,
            "max_length": 65535,
            "example": ""
        }]
    }],
    "default_values":
    {
        "transformScriptContent": "/**\n * @description Default functions executed by the conversion node
\n",
        "outputScriptContent": "importClass(com.bsi.utils.HttpUtils);}"
    }
  },
  "point_meta":
  {
    "property_mapping_enabled": false,
    "point_predefined": true,
    "config_items": [
    {
        "key": "period",
        "name": "Collection period",
        "description": "Collection period",
        "data_type": "int",
        "required": true,
        "crypted": false,
        "max_length": 65535,
        "example": 10000
    },
    {
        "key": "input_path",
        "name": "Input path",
        "description": "Input path",
        "data_type": "string",
        "required": true,
        "crypted": false,
        "max_length": 65535,
        "example": 10000
    },
    {
        "key": "output_path",
        "name": "Output path",
        "description": "Output path",
        "data_type": "string",
        "required": true,
        "crypted": false,
        "max_length": 65535,
        "example": 10000
    }],
    "default_points": [
    {
        "point_id": "material_sync",
        "name": "Material synchronization",
        "data_type": "object",
        "collection_config":
        {
            "period": 5,
            "input_path": "/workingPlan20",
            "output_path": "/opendata/v1/erp/production/plan",
            "transformScript": "transformScriptContent",
```

```
            "inputScript": "inputScriptContent",
            "outputScript": "outputScriptContent"
        }
    },
    {
        "point_id": "order_sync",
        "name": "Order synchronization",
        "data_type": "object",
        "collection_config":
        {
            "period": 5,
            "input_path": "/workingPlan20",
            "output_path": "/opendata/v1/erp/production/plan",
            "transformScript": "transformScriptContent",
            "inputScript": "inputScriptContent",
            "outputScript": "outputScriptContent"
        }
    }]
  }
}
```

Table 2-31 IT data collection template

Key	Type	Description
tpl_id	String	Data collection template ID, which is unique for a tenant. Pattern: '^[a-zA-Z0-9_-]*$' Value length: 1–64 characters
name	String	Data source template name (English). Value length: 1–64 characters
description	String	Description of the data source template. The value can contain 0 to 128 characters.
datasource_meta	Object	Data source configuration metadata.
point_meta	Object	Scenario collection configuration metadata.

Table 2-32 datasource_meta

Key	Type	Description
config_tabs	Array	Data source configuration table list.

Key	Type	Description
default_values	Object	Default data source value, which is usually used to describe the default driver protocol.

Table 2-33 config_tabs

Key	Type	Description
key	String	Key of the data source configuration table. Options include **connection_info** and **collection_paras**. Configure the connection information fields of the IT system in **connection_info**. Configure the fields of the IT system data collection task in **collection_paras**.
name	String	Name of the data source configuration table. The value can contain 1 to 64 characters.
description	String	Description. The value can contain 0 to 255 characters.
config_items	Array	Configuration item list, which is generated on the GUI for configuring **connection_info** or **collection_paras**.

Table 2-34 default_values

Key	Type	Description
key_name	String	The key and value can be customized. A conversion script can be configured. A JavaScript script is used as an example.

Table 2-35 point_meta

Key	Type	Description
property_mapping_enabled	Boolean	Whether to enable property mapping. Set this parameter to **false** for IT data collection.
point_predefined	Boolean	Whether to define points. Set this parameter to **true** for IT data collection.
config_items	Array	Configuration item list, which is generated on the GUI. It applies to all **default_points** values. Each **default_point** value can be configured separately, and the fields with the same key in **collection_config** will be overwritten.
default_points	Array	List of collection task scenarios.

Table 2-36 config_items

Key	Type	Description
key	String	Configuration item key. The value can contain 1 to 32 characters.
name	String	Default item name. The value can contain 1 to 64 characters.

Key	Type	Description
description	String	Configuration item description. The value can contain 0 to 128 characters.
data_type	String	Configuration item type. The value can be **short**, **ushort**, **int**, **int64**, **uint**, **long**, **ulong**, **float**, **double**, **bool**, **string**, **object**, and **decimal**.
required	Boolean	Whether a configuration item is mandatory.
crypted	Boolean	Whether a configuration item is encrypted.
max_length	Integer	Maximum length of an input string. This parameter is valid only when **data_type** is set to **string**.
example	String	Example of a configuration item. The value is displayed in gray in the text box on the GUI. The value can contain 0 to 256 characters.

Table 2-37 default_points

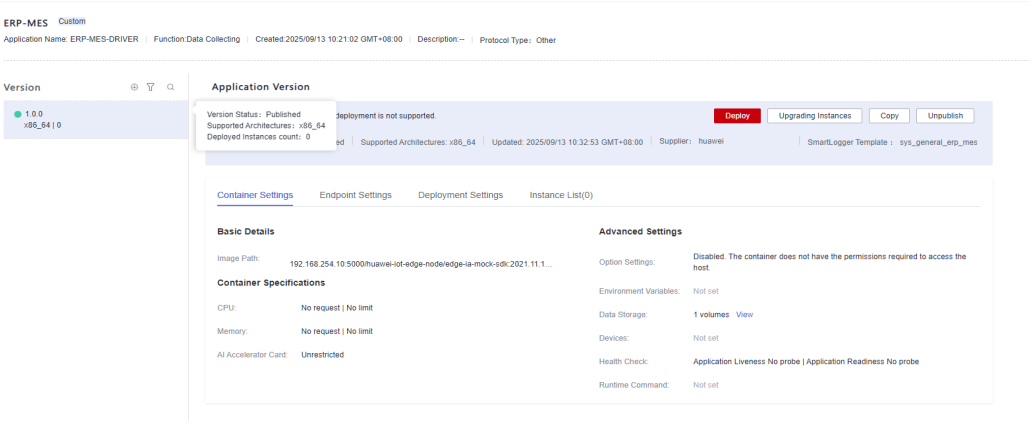
Key	Type	Description
point_id	Boolean	Collection scenario ID.
name	Boolean	Collection scenario name.
data_type	String	Scenario data type. The value is **Object** for IT data collection.
collection_config	Object	Default configuration of the collection scenario.

Step 2 Choose **IoTEdge** > **Applications**, click the **Driver Applications** tab, and click **Add Application**. Set the protocol type to **HTTP** or **JDBC** based on the data collection

mode. If multiple modes are used, select **Other**. Set the function to data collection.

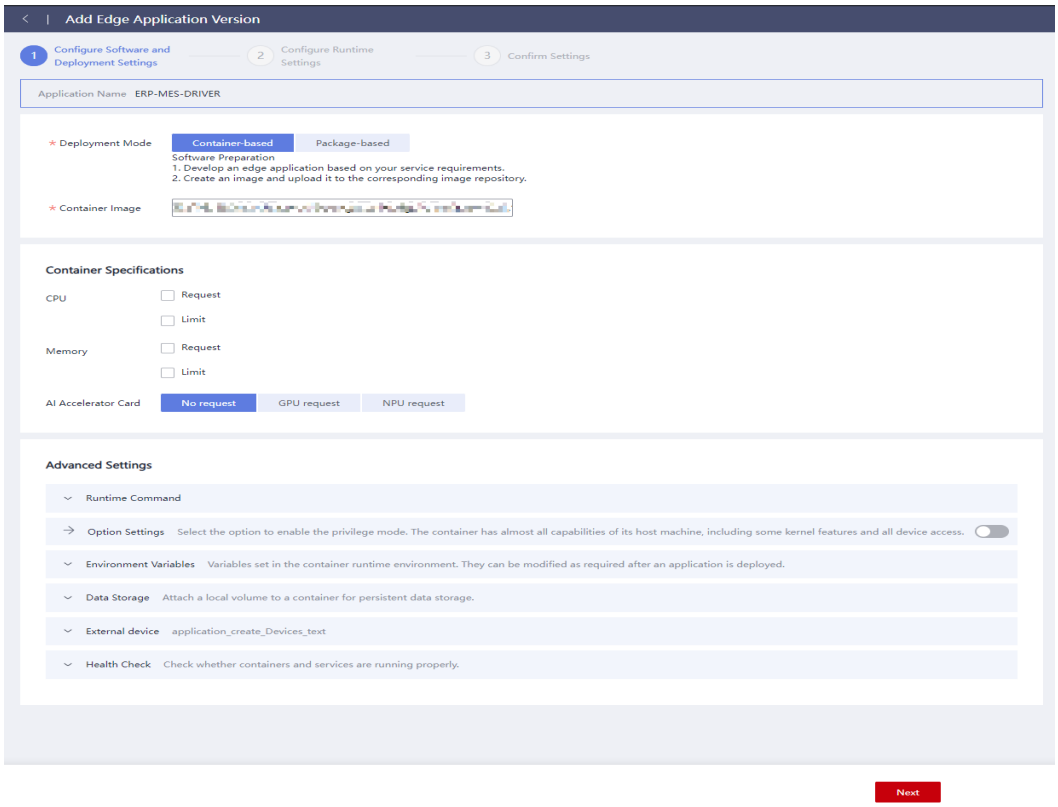
Step 3 Choose **IoTEdge** > **Applications**, click the **Driver Applications** tab, click the created driver application, and click the plus sign (+) to create an application version.

Figure 2-110 Creating a driver application version



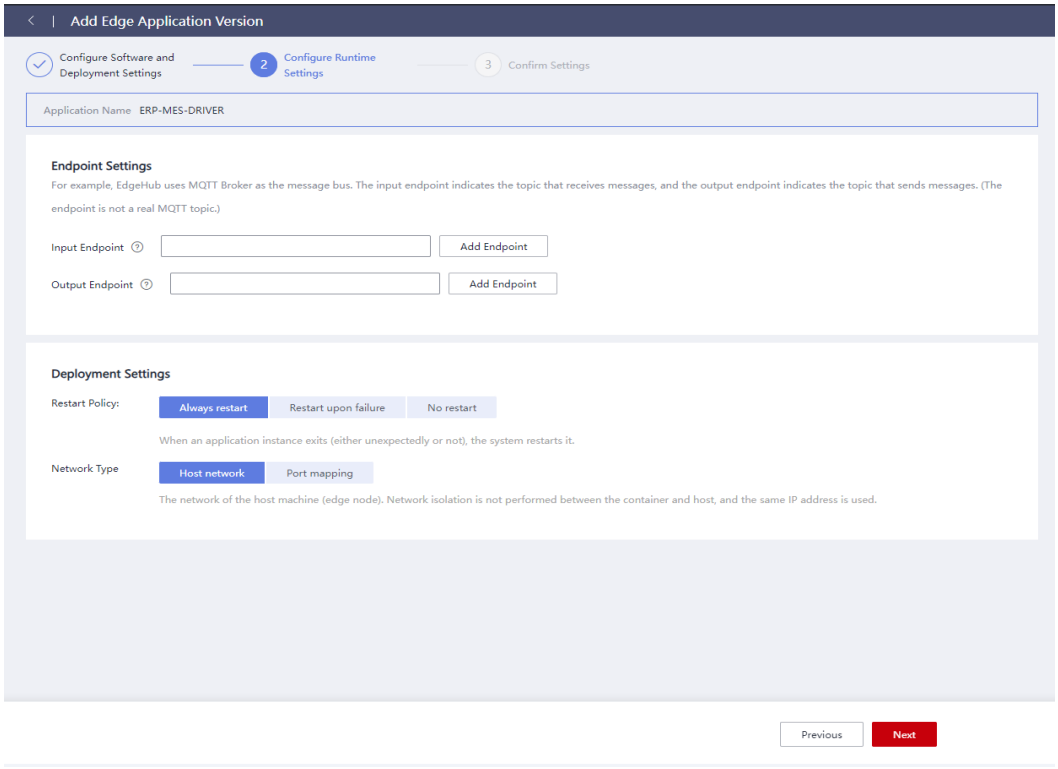
Select **Container-based** deployment, enter the container image path, and click **Next**. Based on your IT data collection service requirements, develop driver applications, create images, and upload the images to an image repository.

Figure 2-111 Software and deployment settings



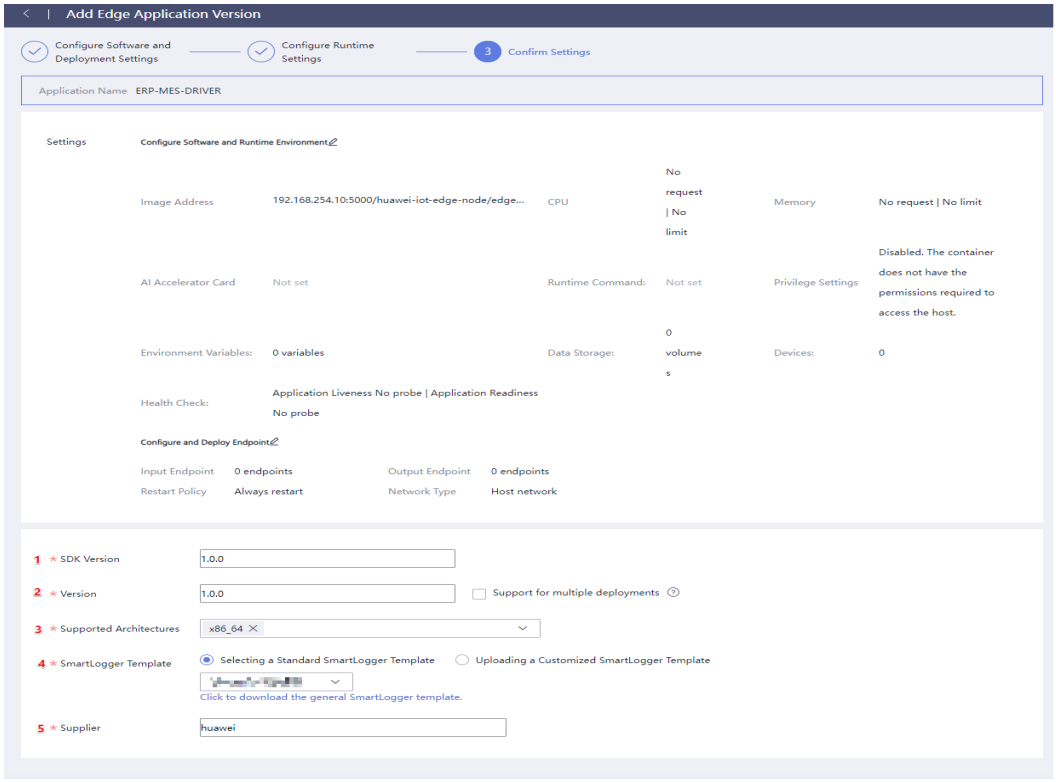
Enter the runtime configuration based on the application requirements. If it is not required, click **Next**.

Figure 2-112 Runtime configuration



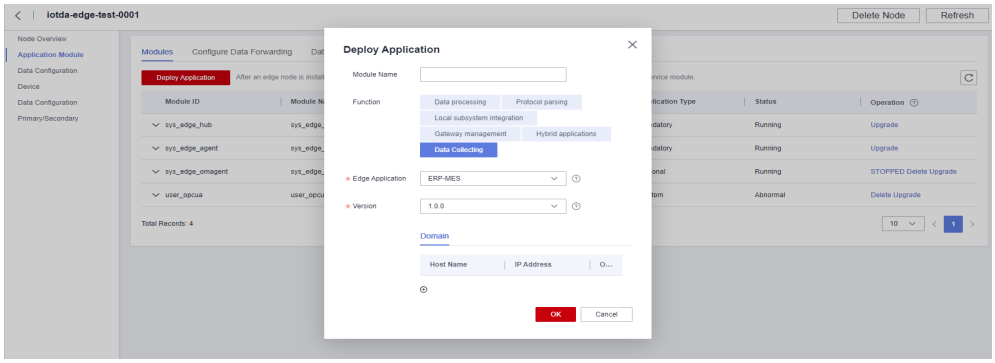
Enter the integrated SDK version, application version, and supported architecture. Bind the added data collection template. Enter the vendor name and click **Publish Now**.

Figure 2-113 Confirming configuration



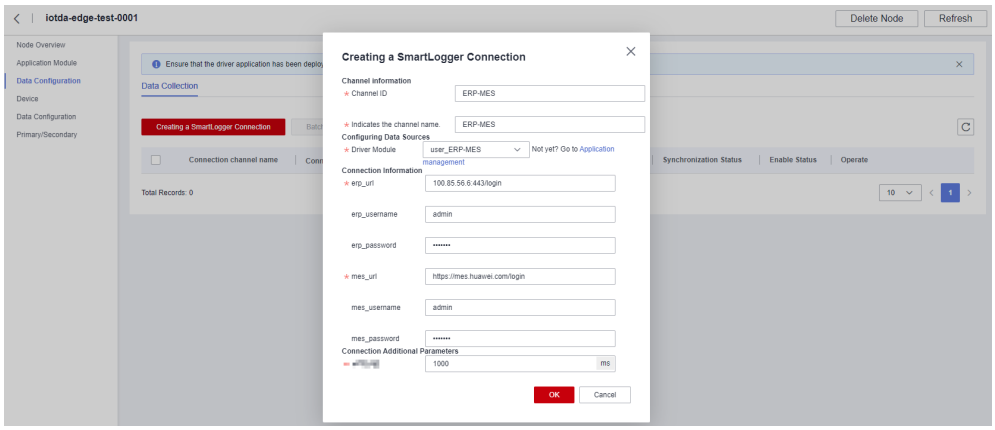
Step 4 Go the edge node page, choose **Application Module** in the navigation pane, click the **Modules** tab, and click **Deploy Application**. In the dialog box displayed, select data collection, the created application, and application version, and click **OK**.

Figure 2-114 Deploying an application



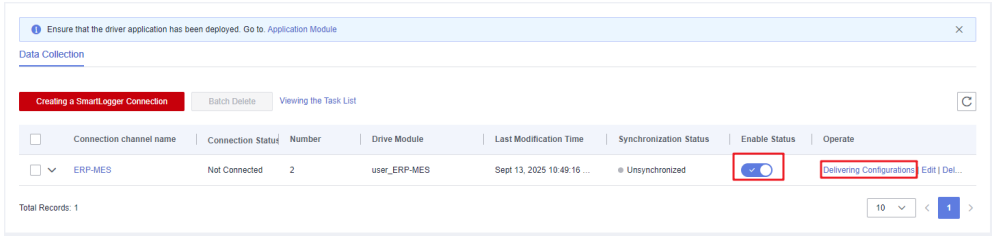
Step 5 Choose **Data Configuration** in the navigation pane, click the **Data Collection** tab to create a data collection connection. In the dialog box displayed, select the created application module. Enter IT system information and click **OK**.

Figure 2-115 Creating a data collection connection



Step 6 Enable the created data collection connection. Deliver the data collection connection information and the collection task predefined in the data collection template to the created application.

Figure 2-116 Delivering configuration



----End

2.11.1.3.5 Batch Task Import

Creating a Task for Adding Points in Batches

Step 1 Log in to the IoTDA console. In the navigation pane, choose **IoTEdge** > **Nodes**. On the page displayed, click the created edge node to access its details page.

Step 2 Choose **Data Configuration**, click the connection channel, import points, and add them in batches. You can use incremental import or full import. Incremental import is to update existing points and insert new points. Full import is to delete existing points and insert new points.

Step 3 Enter a task name, download the Excel template file, add point data, and upload the file.

Table 2-38 Point template file description

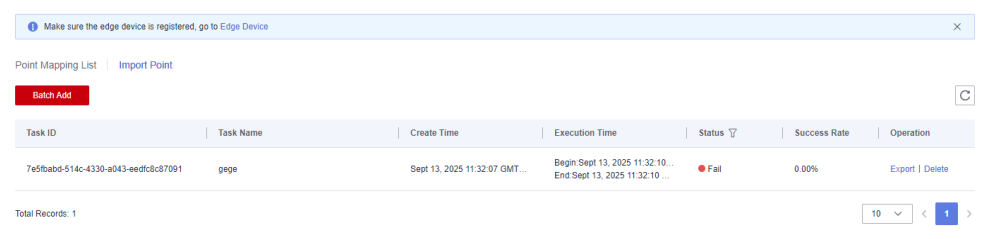
Parameter	Description
common_config.point_id	Point ID.

Parameter	Description
common_config.name	Point name.
common_config.device_id	Device ID.
common_config.property	Property path.
common_config.data_type	Data type.
collection_config.address	Point address.
collection_config.data_type	Data type of point value.
collection_config.cycle	Point collection period.
stream_config.stream_formula	• Formula for complex mapping. Currently, **bit()** and **bool()** are supported. They support integers only. • Bit splitting: For example, **bit(0)** indicates that only bit 0 of the collected point value is obtained and the value is reported as an integer. For example, if the collected point value is 3 (its binary value is 0000 0011) and **bit(0)** is configured, **1** will be reported. • **bool()** is an enhanced operation of bit splitting. It converts an integer to a Boolean value, for example, **bit(0).bool()**. For example, if the collected point value is 3 (its binary value is 0000 0011) and **bit(0).bool()** is configured, **true** will be reported.
scaling_config.ratio	Scaling ratio.
scaling_config.base	Baseline value.
scaling_config.accuracy	Precision of the scaling result. **-1** (default value) indicates that all decimal places are retained. **0** indicates that only integer places are retained. **1** indicates that only one decimal place is retained. If this parameter is left empty, the default value is used.
clean_config.silent_window	Silence time window. If no reporting condition is triggered within this time window, the point will not be reported.
clean_config.deviation	Deviation. Values within this range are considered normal fluctuations and are not reported.
validity_config.min	Minimum point value for reporting. If a value is less than the minimum value, an alarm is reported.

Parameter	Description
validity_config.max	Maximum point value for reporting. If a value is greater than the maximum value, an alarm is reported.

Step 4 The batch task is successful.

Figure 2-117 Batch tasks



NOTE

1. Up to 20 batch tasks can be created in a resource space.
2. Up to 10 batch tasks can be processed in a resource space at a time.
3. Each batch task, including batch device registration and deletion, is automatically aged and deleted after 30 days.
4. Up to 10,000 subtasks can be added to a batch task.
5. During batch point registration, if a data source has tasks that are waiting or being executed, new tasks cannot be created.
6. The task that is being executed cannot be deleted.
7. A scheduled batch task is executed every 5 minutes.
8. If no result is displayed in the exported Excel file but the task execution fails, the possible cause is that the data source or node may be deleted but the task still exists.

----End

2.11.1.3.6 Child Devices

Edge Devices

Choose **IoTEdge** > **Nodes**, click the edge node name to go to the node details page, and choose **Device** on the left. You can view or add edge devices, configure, delete, and manage child devices, as well as register MQTT edge devices in batches.

For details on device access, see **Connecting a Device to an Edge Node**.

Figure 2-118 Child device management

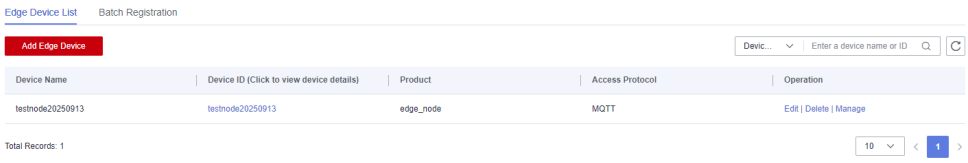


Table 2-39 Operations

Parameter	Description
Edit	Check or modify the configuration of an edge device.
Delete	Delete an edge device.
Manage	View details about the edge device and register a child device. For details, see **Registering a Modbus Child Device** or **Registering an OPC UA Child Device**.

NOTE

When creating an MQTT device under an edge node on the IoTDA platform, do not set the module ID parameter. Otherwise, MQTT authentication fails.

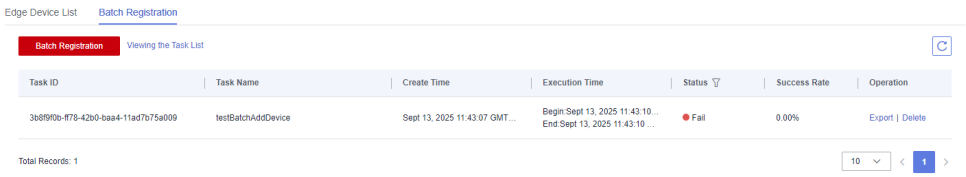
Registering MQTT Edge Devices in Batches

The platform allows you to add MQTT edge devices in batches.

Step 1 Choose **IoTEdge** > **Nodes**, click your edge node, choose **Device** on the left, and click the **Batch Registration** tab.

Step 2 Click **Batch Registration**, download the template file to the local host, enter device information based on the parameter description in the file, upload the file to the platform, and click **OK**. The system automatically generates a batch device registration task. The task status will be available in about 3 minutes.

Figure 2-119 Batch registration



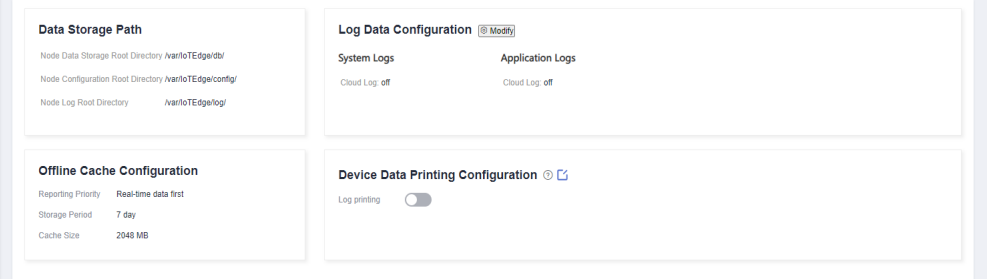
----End

2.11.1.3.7 Data Configuration

Step 1 Choose **IoTEdge** > **Nodes**, and click your edge node.

Step 2 Choose **Data Configuration**. Check the data storage path, offline cache configuration, and log configuration of the edge node, and modify the log configuration.

Figure 2-120 Log settings



----End

2.11.1.3.8 Remote Maintenance

IoTEdge provides a web-based client tool to enable you to use SSH to remotely access nodes.

Prerequisites

- The node is not in the uninstalled, upgrading, or deleting state.
- An SSH server program is installed on the remote node, and listening to port 22 is enabled.

CAUTION

- To remotely log in to a node, you must enter the correct username and password.
- If the node status is **Offline**, the connection may fail to be established. Consequently, remote maintenance is unavailable.
- Only one user can log in to the node at a time. A new login request will forcibly log out the existing user.
- Tab switching will close the connection. If this happens, you must log in again.
- If the version of $edge_agent is later than 1-1-6, the remote SSH function is not supported. To use the remote SSH function, deploy $edge_omagent 1-1-11 or later.

Procedure

Step 1 Log in to the IoTDA console.

Step 2 Choose **IoTEdge** > **Nodes** in the left navigation pane.

Step 3 Click the target edge node name to access its details page.

Step 4 Click the **Remote Maintenance** tab, and then click **Log In**.

Step 5 In the dialog box displayed, enter the username and password used for login, and click **OK**.

Step 6 If the username and password are correct, the system displays a message indicating that the remote login is successful. Expand the directory tree.

Figure 2-121 Successful login

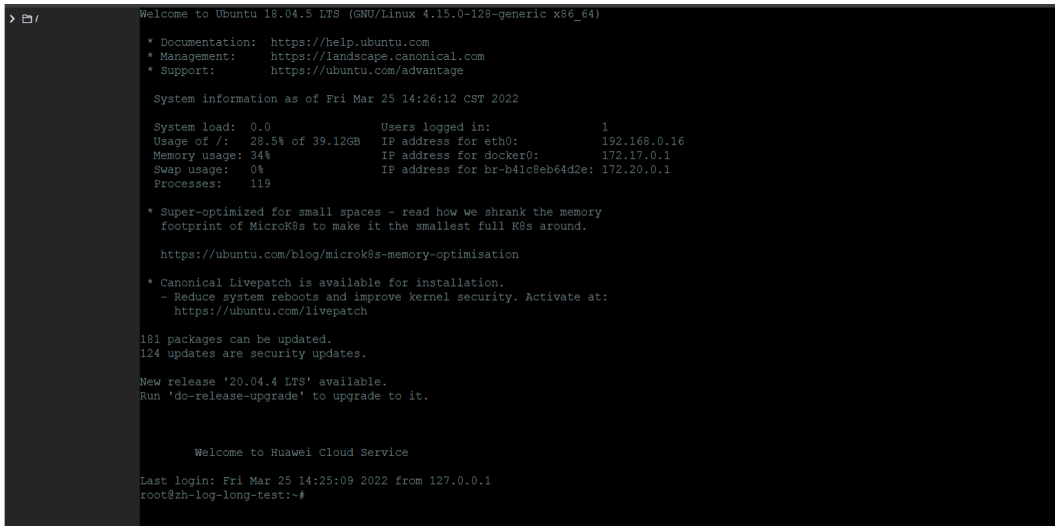
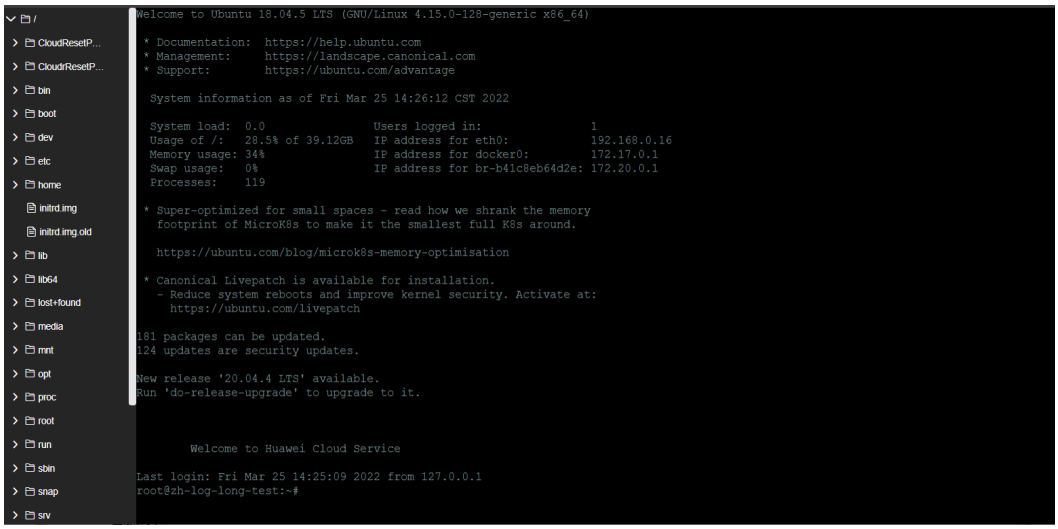


Figure 2-122 Expanding the directory tree

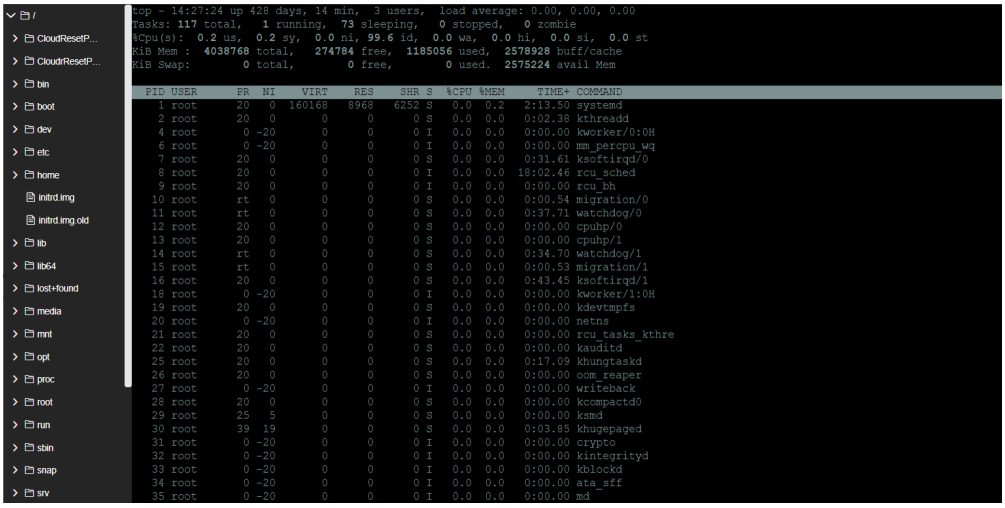


Step 7 Interact with the node.

For example:

1. Run the following command to check the system running status:
   > **top**

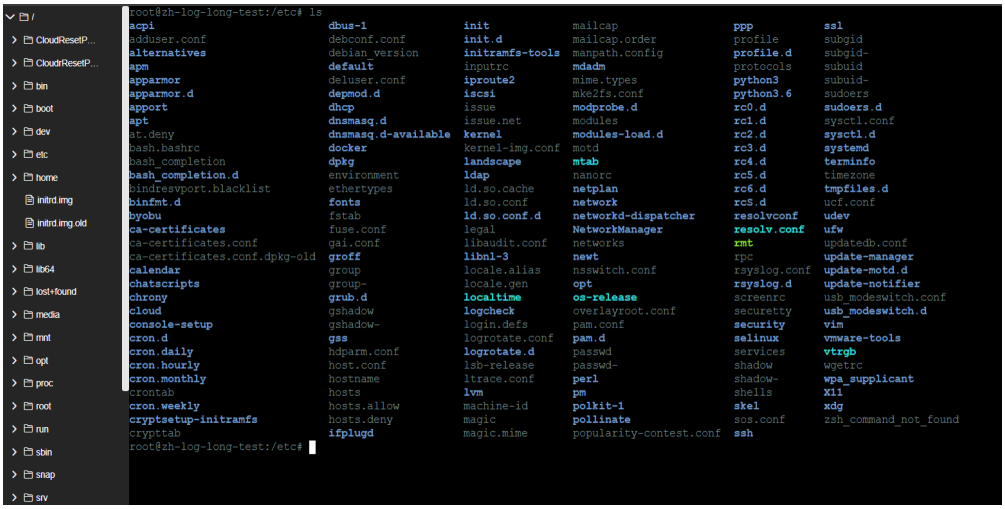
Figure 2-123 top command output



2. Run the following command to check files and directories in the **/etc** directory:

   > **ls**

Figure 2-124 ls command output

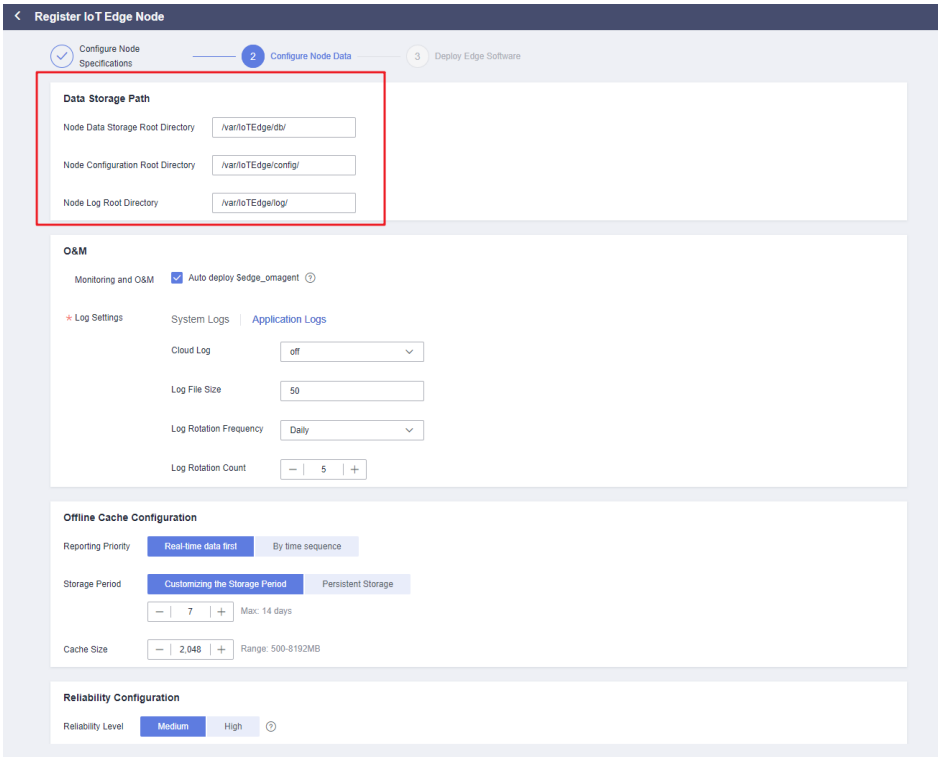


Step 8 Upload and download a file.

CAUTION

1. Only files can be downloaded.
2. Files can be uploaded only to folders. It indicates that files are uploaded to the corresponding folders on the remote host.
3. The maximum size of a file to be uploaded or downloaded is 10 MB.
4. The target directory for uploading and downloading a file is restricted by the root directory configured during node creation.

Figure 2-125 Configuring node data



5. To upload and download a file, you need to deploy $edge_omagent 1-1-11 or later.

Right-click the target folder or file. The file upload and download options are displayed.

Figure 2-126 Uploading and downloading a file

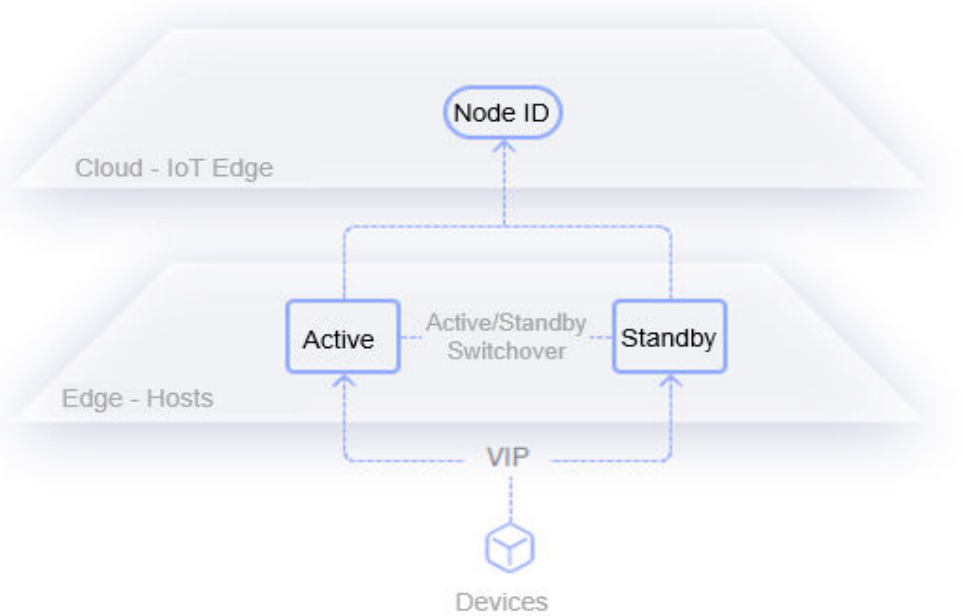
Step 9 Log out of the node.

Close the browser, refresh the page, click the close button in the upper right corner, or run the **exit** command on the operation page to log out of the node.

----End

2.11.1.3.9 Active/Standby Configuration

Figure 2-127 Active/Standby Configuration



The active/standby configuration ensures the reliability of active node data and the continuity of related services.

The active node executes all edge services. The standby node downloads all images and executes only the edge_agent and edge_keepalived applications. When the active node breaks down or the network is faulty, the active/standby

switchover is performed. After the original active node recovers, it does not perform preemption.

Constraints

1. To configure the active/standby mode, the network interface cards (NICs) on the bound gateway must be on the same network to ensure normal heartbeat.
2. Correct **iptables** must be configured between the active and standby gateways to prevent heartbeat network exceptions.
3. You are advised to check the firewall configuration. If the firewall is not disabled, add the Virtual Router Redundancy Protocol (VRRP) bypass policy.
4. To delete the active/standby configuration, uninstall the standby server that is installed later by running the **sh /opt/IoTEdge-Installer/uninstall.sh** command. Then, delete the active/standby configuration and retain the first installed host. If the host that is installed later is not uninstalled, deleting the active/standby configuration will stop the application by default, affecting node running.
5. If the NIC name is incorrect, the **edge_keepalive** module will be faulty. As a result, the active and standby nodes cannot work properly.
6. Currently, the **edge_keepalive** upgrade will cause an active/standby switchover. Therefore, you are not advised to upgrade it.

Prerequisites

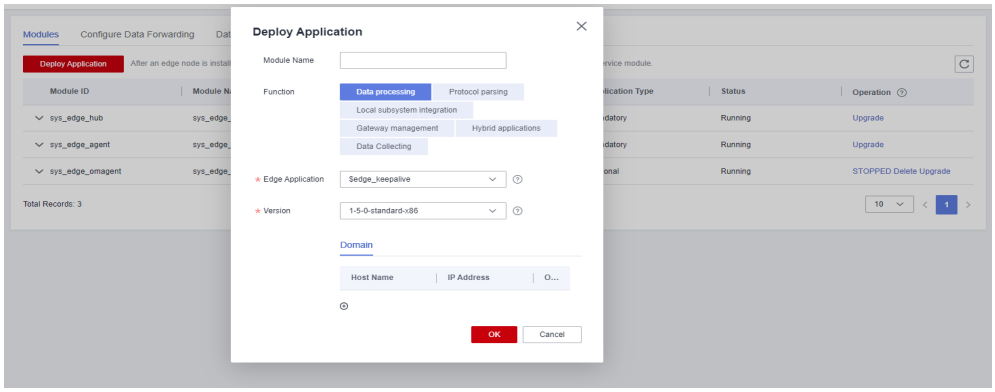
Register and install edge nodes. For details, see **Registering an Edge Node** and **Installing an Edge Node**.

Deploying an Application

Step 1 Click the target edge node name to access its details page.

Step 2 In the navigation pane on the left, choose **Application Module**, and click the **Modules** tab, and click **Deploy Application** to deploy the $edge_keepalive application.

Figure 2-128 Deploying an Application

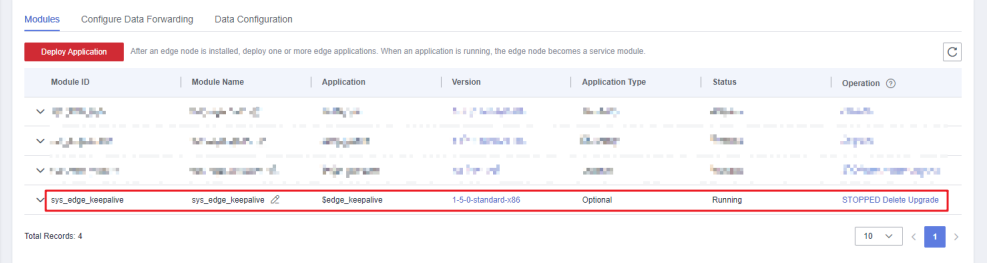


- **Edge Application**: Select **$edge_keepalive**.

- **Version**: Select **1-1-40-standard-x86**. (Set it to the actual version in the current environment.)

Step 3 In the navigation pane on the left, choose **Application Module**, click the **Modules** tab to view the deployed edge application.

Figure 2-129 Checking deployed edge applications



----End

Adding the Active/Standby Configuration

Step 1 Click the target edge node name to access its details page.

Step 2 In the navigation pane on the left, choose **Primary/Secondary** and click **Active/Standby Configuration**.

Step 3 Enter the information and click **OK**.

Figure 2-130 Active/Standby Configuration

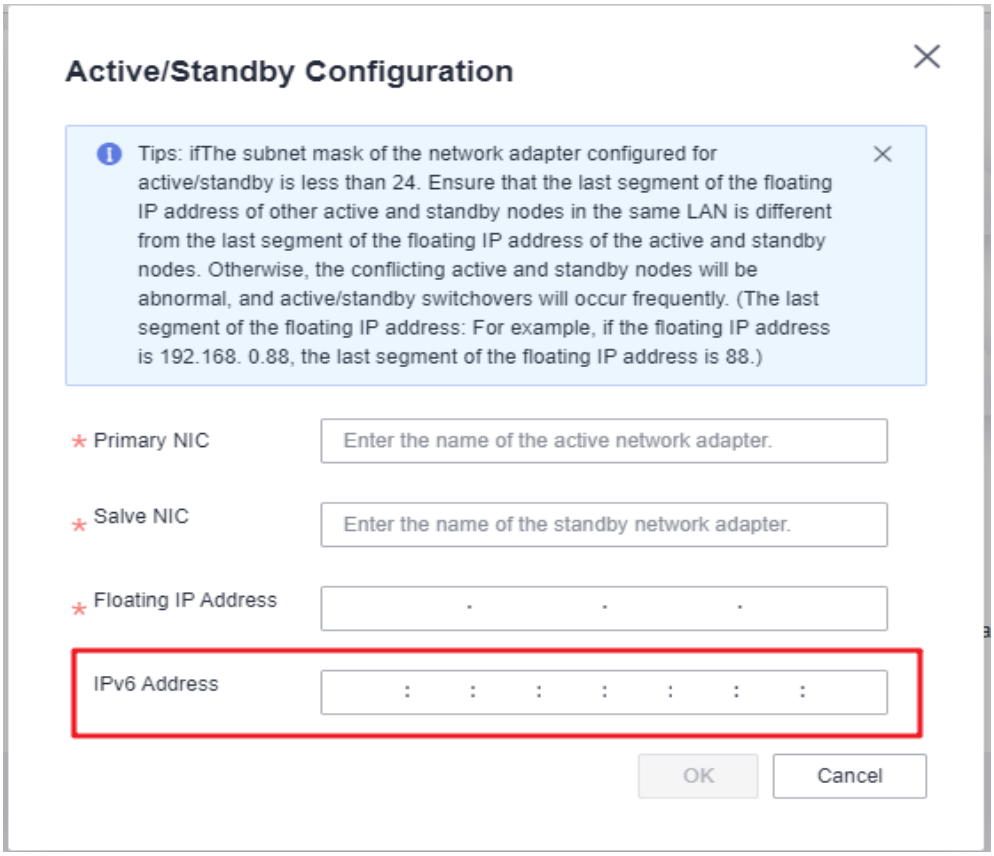
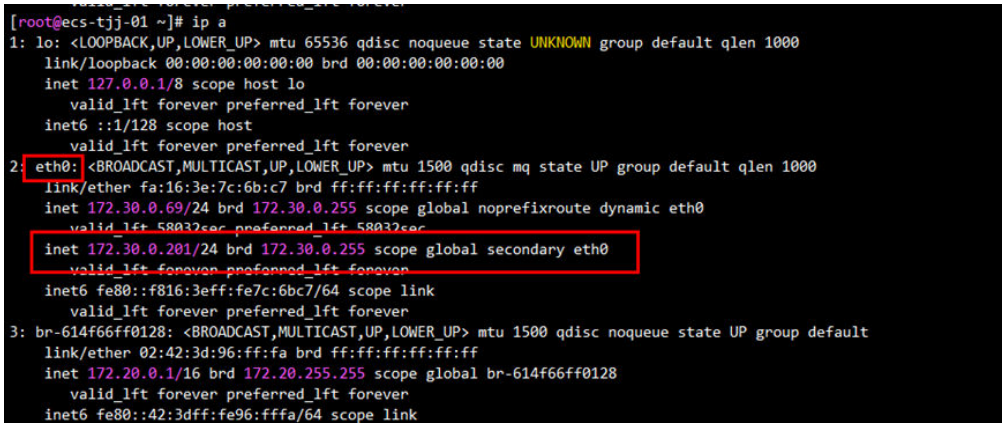


Table 2-40 Active/Standby configuration

Parameter	Description
Active NIC	Name of the NIC bound to the virtual IP address on the active node, for example, **eth0** and **eth1**. Ensure that the name is correct. Otherwise, the virtual IP address will fail to be bound.
Standby NIC	Name of the NIC bound to the virtual IP address on the standby node, for example, **eth0** and **eth1**.
Floating Virtual IP Address	Virtual IP address for external systems. It is used for device connection. Set it based on your service plan.

Step 4 View the IP address information. Run the **ip a** command to view the IP address of the active node. If the current settings are correct, the bound virtual IP address can be retrieved on the active node. For example, the current virtual IP address is **172.30.0.201**, and the NIC bound to the active node is **eth0**.

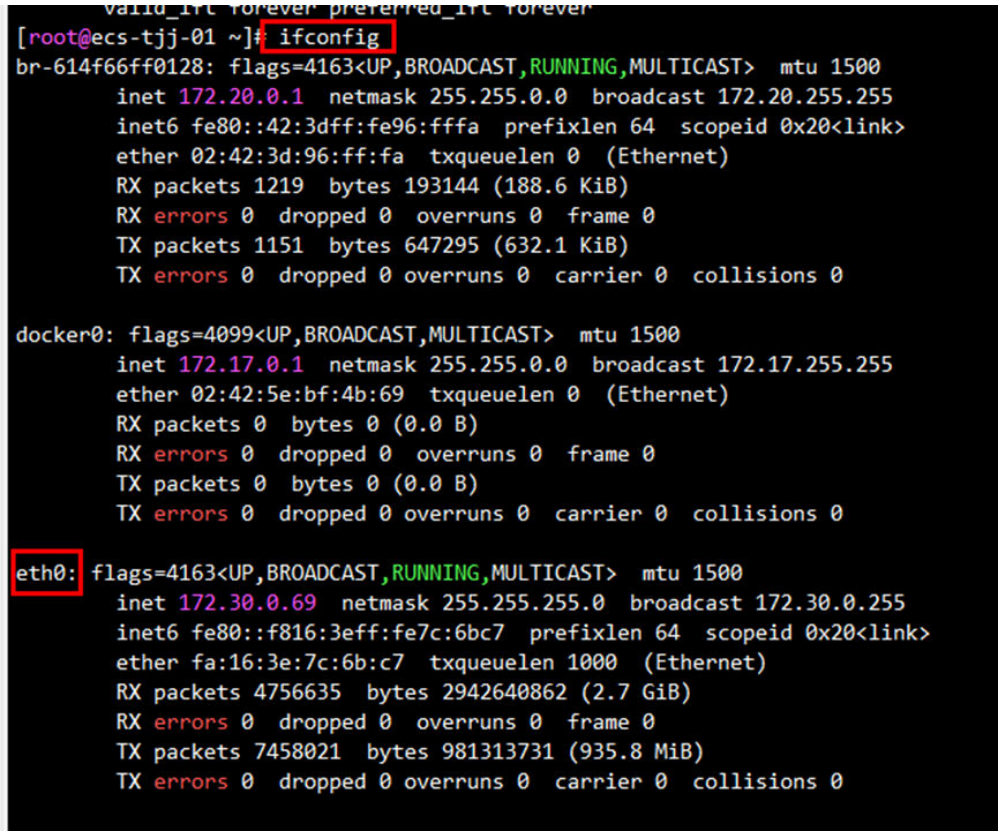
Figure 2-131 Querying IP address information



NOTE

Run the **ifconfig** command to view the NIC name, as shown in the following figure.

Figure 2-132 Checking the NIC name



----End

Installing a Standby Node

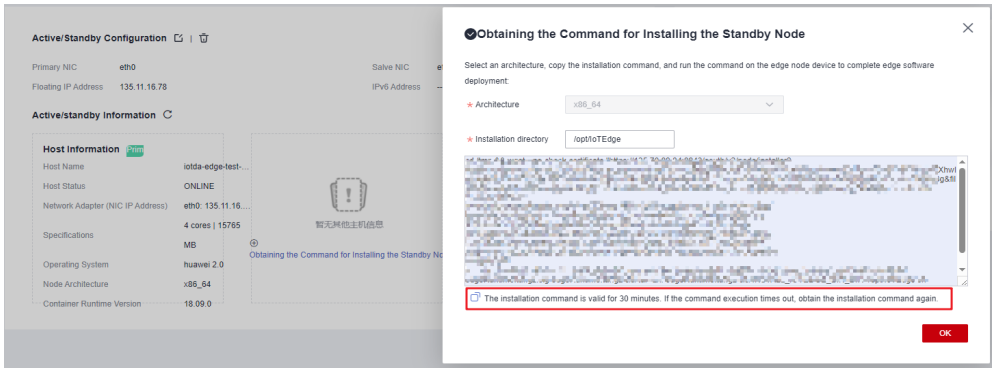
Step 1 Click the target edge node name to access its details page.

Step 2 In the navigation pane on the left, choose **Primary/Secondary** to obtain the command for installing the standby node.

CAUTION

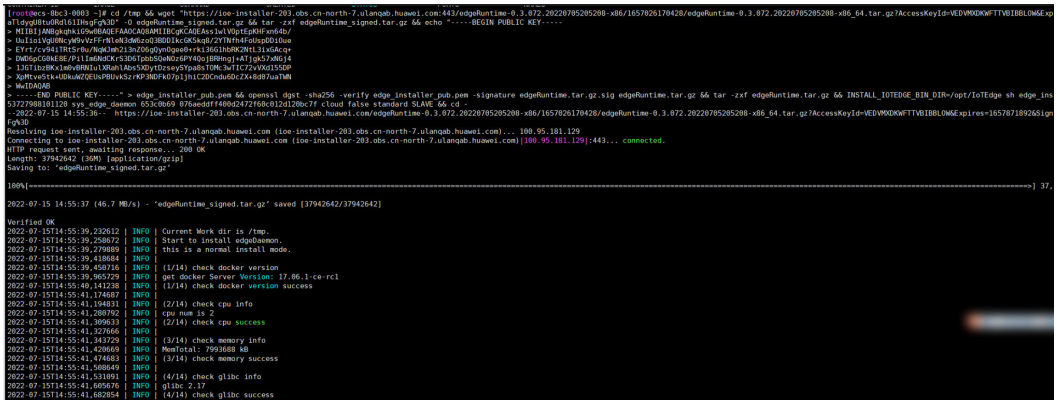
The network between the bound NICs on the active and standby nodes must be normal. Otherwise, the active and standby nodes may be abnormal.

Figure 2-133 Obtaining the installation commands of the standby node



Step 3 Log in to the standby edge node and run the installation command.

Figure 2-134 Running the installation commands



If the following information is displayed, the standby node is installed.

Figure 2-135 Installation successful

By default, only the edge_agent and edge_keepalive modules run on the standby node. Statuses of other modules are **Created**.

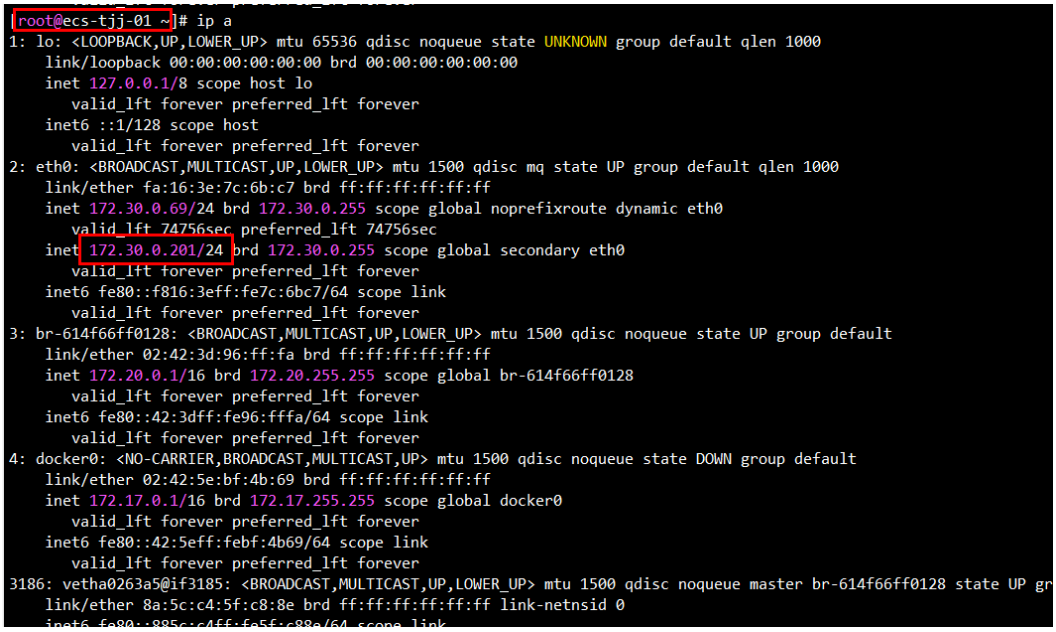
Figure 2-136 docker ps command output

----End

Checking the Binding Status of the Current Virtual IP Address

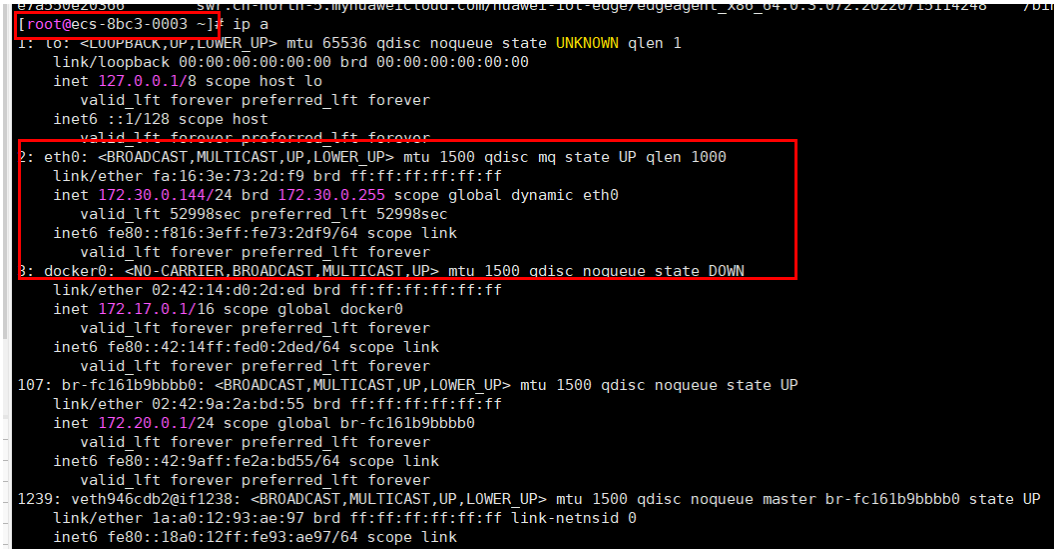
Check the active node. The **eth0** NIC on the active node is bound to a virtual IP address and the node is a working node.

Figure 2-137 Checking the active node



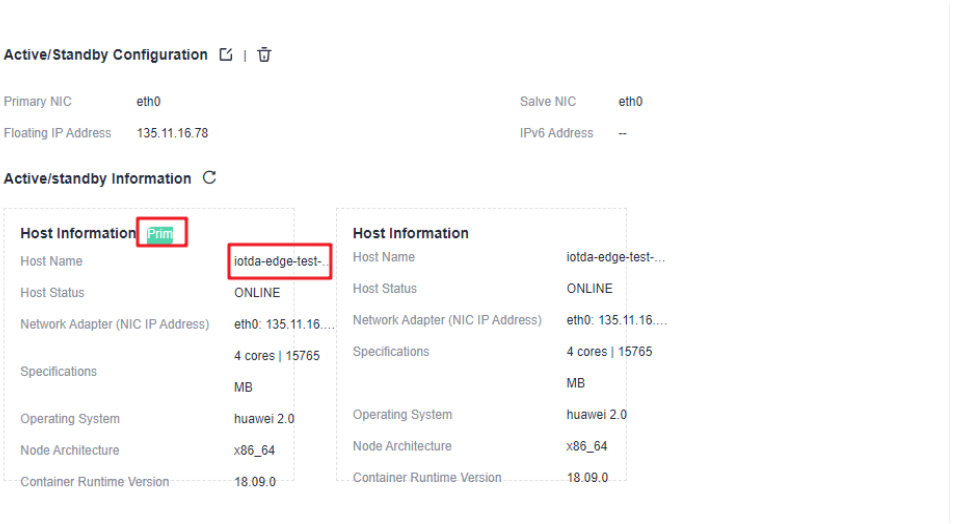
Check the IP address list of the standby node. The **eth0** NIC is not bound to a virtual IP address.

Figure 2-138 Checking the standby node



Check the active and standby nodes on the console. The working node is **ecs-tjj-01**. (The green label indicates that the node is a working node.)

Figure 2-139 Checking the active/standby configuration node



The active/standby configuration is completed.

Simple Test

Simulate a fault on the current working node so that heartbeat packets are not sent to the standby node.

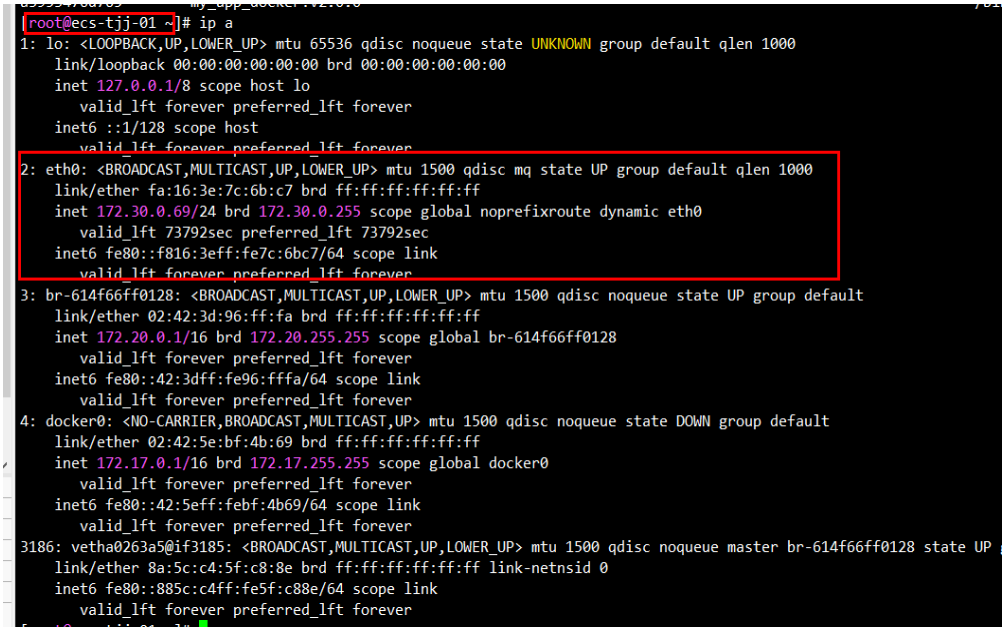
- Stop keepalive on the active node to simulate a breakdown.

  Figure 2-140 Simulating a breakdown

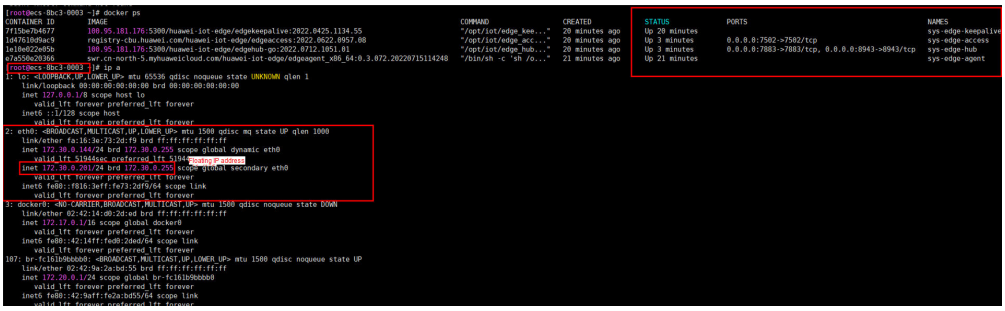
  After the application is stopped, check the application status. The hub status is **Created**. Check the floating IP address. It is found that the virtual IP address of the original active node has been removed.

Figure 2-141 Virtual IP address of the original primary node removed



- Check the floating IP address of the standby node and the module status.

Figure 2-142 Querying the module running status of the standby node



The floating IP address has been bound to the **eth0** NIC on the standby node, and all service modules are running properly on the original standby node.

- Check the active/standby node information on the console. The working node has been switched to the **ecs-8bc3-0003** standby node.

Figure 2-143 Checking the active/standby configuration node

The simple test of the active/standby switchover is completed.

2.11.1.3.10 Deleting an Edge Node

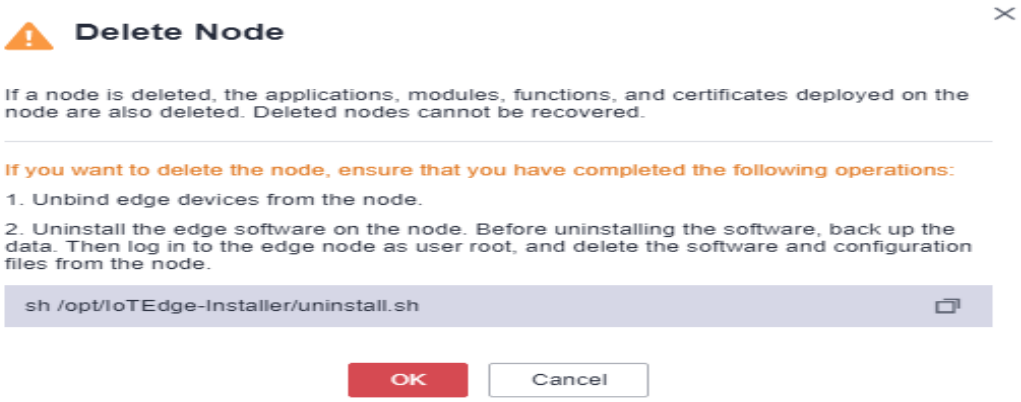
Delete an edge node if it is no longer needed.

Procedure

Step 1 Choose **IoTEdge** > **Nodes** in the left navigation pane. All edge nodes are displayed.

Step 2 Select the target edge node and click **Delete** in the **Operation** column. Read the message in the dialog box displayed.

Figure 2-144 Confirm



Step 3 (Optional) Unbind edge devices from the edge node.

NOTE

You can delete an edge node only after all devices bound to the edge node are deleted.

Step 4 Uninstall the edge software on the edge node.

Copy the command displayed in the dialog box, use the SSH tool to log in to the background system of the edge node, and run the command as user **root** to delete the software and configuration files on the edge node.

Step 5 Click **OK** and wait until the edge node is deleted.

----End

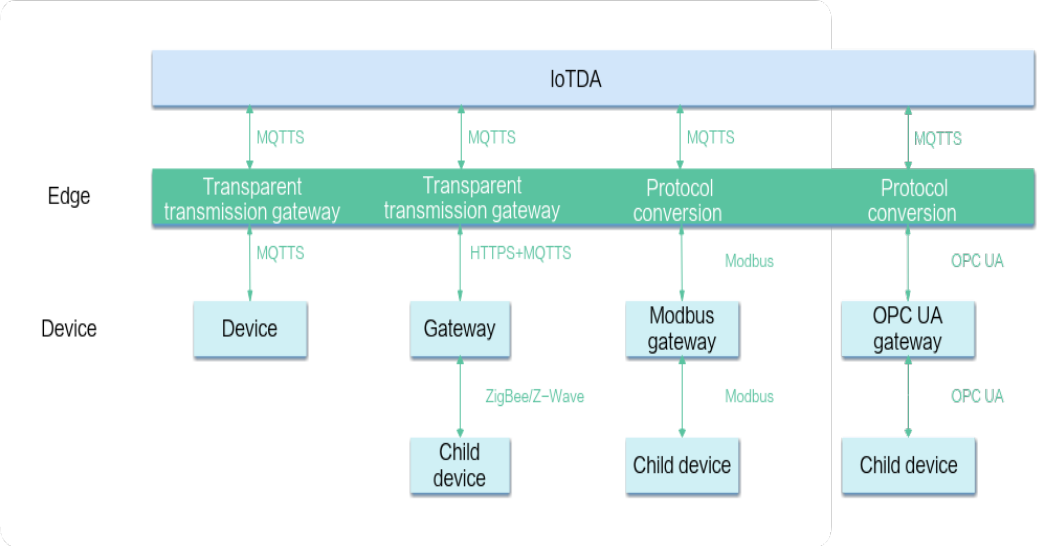
2.11.2 Connecting a Device to an Edge Node

2.11.2.1 Connection Mode

After an IoTEdge application is deployed, the edge node functions as an extension of IoTDA on the device side to manage devices through cloud-edge synergy. The edge node provides computing and management services for nearby devices, such as local management of low-latency services and local control upon disconnection from IoTDA. The devices connect to the edge node and report data to IoTDA through the edge node. The edge node supports two connection modes:

- **Protocol conversion**: The edge node supports access using Modbus and OPC Unified Architecture (OPC UA). It converts Modbus and OPC UA messages into MQTTS messages and reports data to IoTDA in JSON format. Devices using Modbus or OPC UA are treated as child devices of the edge node.
- **Transparent transmission gateway**: The edge node supports MQTTS access. MQTT devices or gateways can connect to IoTDA directly or through edge nodes. The edge node functions as a gateway that transparently transmits data reported by devices or gateways to IoTDA.

Figure 2-145 Connection modes



The table below describes the two connection modes.

Connection Mode	Applicable Device Type	Scenario
Protocol conversion	Devices that use Modbus or OPC UA to connect to edge nodes	Edge nodes function as gateways, and devices connect to IoTDA as child devices of the edge nodes. The devices send data to the edge nodes. The edge nodes convert the data format into JSON, encapsulate the JSON data into MQTTS messages, and report the MQTTS messages to IoTDA.
Transparent transmission gateway	Devices that use MQTTS to directly connect to edge nodes	Devices integrate AgentLite SDK, AgentTiny SDK, or LiteOS or implement the native MQTTS protocol. They send data to edge nodes, which then transparently transmit the data to IoTDA.
	Devices that do not support the TCP/IP protocol stack, cannot directly communicate with IoTDA, and want to use gateways to connect to edge nodes for near-end management	Devices function as child devices of gateways, the gateways are directly connected to edge nodes, and the edge nodes transparently transmit data to IoTDA. • The gateways connect to the edge nodes by integrating IoT AgentLite SDK. • The devices connect to the gateway through the near-field communication protocol so the devices and gateways use the same protocol.

NOTICE

IoTEdge transparently transmits IoTDA packets. **Table 2-41** lists the capabilities. For devices described in **Gateway requesting for adding child devices**, you can only view their information on the IoTDA console and cannot manage them on the IoTEdge console.

Table 2-41 IoTDA packet transparent transmission capabilities of IoTEdge

Type	Supported
1 Device commands	Yes
1.1 Platform delivering a command	Yes
2 Device messages	Yes

Type	Supported
2.1 Device reporting a message	Yes
2.2 Platform delivering a message	Yes
3 Device properties	Yes
3.1 Device reporting properties	Yes
3.2 Gateway reporting device properties in batches	Yes
3.3 Platform setting device properties	Yes
3.4 Platform querying device properties	Yes
3.5 Device obtaining device shadow data from the platform	Yes
4 Gateway and child device management	Yes
4.1 Platform notifying a gateway of new child device connection	Yes
4.2 Platform notifying a gateway of child device deletion	Yes
4.3 Gateway synchronizing child device information	Yes
4.4 Gateway updating child device statuses	Yes
4.5 Platform responding to a request for updating child device statuses	Yes
4.6 Gateway requesting for adding child devices	Yes
4.7 Platform responding to a request for adding child devices	Yes
4.8 Gateway requesting for deleting child devices	Yes
4.9 Platform responding to a request for deleting child devices	Yes
5 Software/Firmware upgrade	No
5.1 Platform requesting the software or firmware version	No
5.2 Device reporting the software or firmware version	No

Type	Supported
5.3 Platform delivering an upgrade event	No
5.4 Device reporting the upgrade status	No
6 File Upload and Download	No
6.1 Device requesting a URL for file upload	No
6.2 Platform delivering a temporary URL for file upload	No
6.3 Device reporting file upload results	No
6.4 Device requesting a URL for file download	No
6.5 Platform delivering a temporary URL for file download	No
6.6 Device reporting file download results	No
7 Device time synchronization	No
7.1 Device requesting time synchronization	No
7.2 Platform responding to a request for time synchronization	No
8 Device information reporting	No
8.1 Device reporting information	No
9 Device log collection	No
9.1 Platform delivering a log collection notification	No
9.2 Device reporting log content	No
10 Remote configuration	No
10.1 Platform delivering a configuration notification	No
10.2 Device reporting the configuration response	No

2.11.2.2 Protocol Conversion

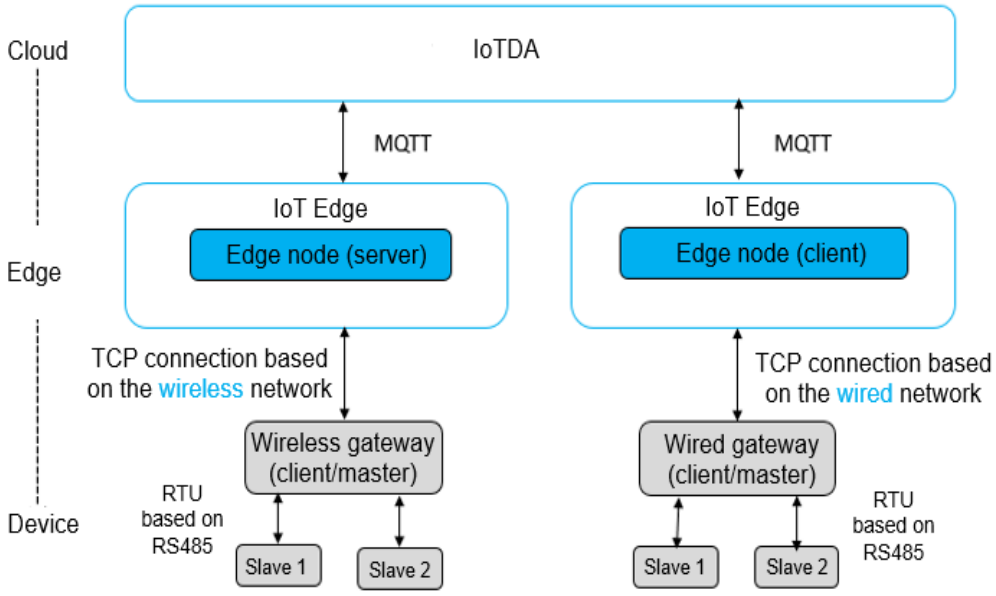
2.11.2.2.1 Modbus Device Access

Overview

Modbus is a communications standard in the industrial field and is the most commonly used connection mode between industrial electronic devices.

Devices using Modbus are not directly connected to IoTEdge nodes.

Figure 2-146 Modbus network topology



Access Mode	Description	Application Scenario
Indirect access	A Modbus device connects to a gateway through the RTU serial port, and the gateway communicates with the edge node using TCP. The gateway functions as the master node, and the Modbus device functions as the slave node. You must bind the Modbus device to the gateway and distinguish Modbus devices connected to the same gateway based on the slave number.	Modbus devices that do not support TCP communication and can communicate only through RTU serial ports

During data collection, the edge node can instruct the gateway to collect binary data from Modbus devices at a specific interval.

After data collection is complete, the edge node normalizes the collected data into JSON format and reports the data to IoTDA using MQTTS.

Connection Procedure

Scenario	Procedure	Description
Device preparation	Prepare a Modbus gateway.	Apply for a gateway that supports Modbus. (Modbus devices can communicate with gateways through RTU serial ports, and the gateways communicate with edge nodes through TCP.)
	Prepare a Modbus device.	Apply for a sensor that supports Modbus.
Platform operations	Develop a product model for the Modbus device.	Develop a product model for the Modbus device on the IoTDA console.
	Add a Modbus edge device (gateway).	Add a Modbus gateway under the edge node.
	Register a Modbus child device.	Register a Modbus child device under the Modbus gateway and bind the child device to the specified Modbus gateway.

Developing a Product Model for a Modbus Device

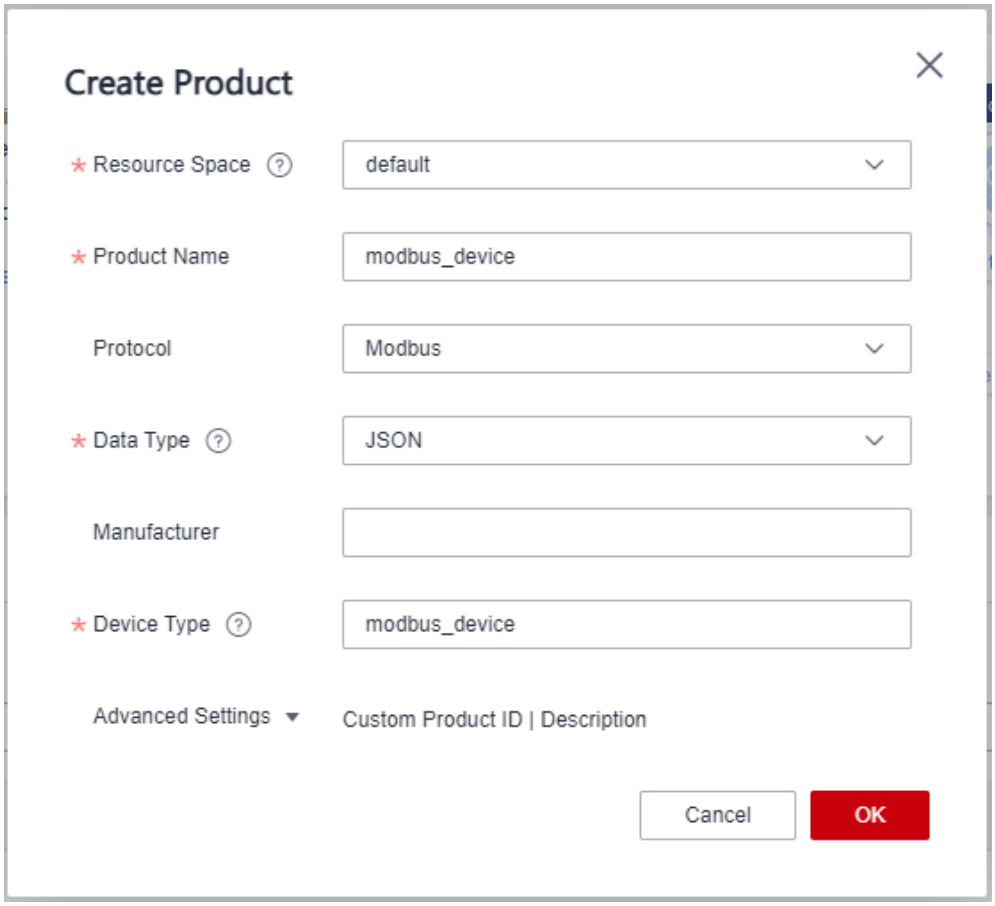
To develop a product model for Modbus device capabilities, **create a product**, define the product model, and build an abstract model for the device on IoTDA, so that IoTDA can understand the services, properties, and commands supported by the Modbus device, such as the temperature and battery level. For details, see "Product Development" > "Developing a Product Model" > "Developing a Product Model Online" in *Developer Guide*.

Create a product whose protocol type is Modbus.

CAUTION

- **Protocol** (mandatory): **Modbus**
- After creating a product, add services and properties for the product. An empty product cannot be used to create a device. For details, see "Product Development" > "Developing a Product Model" > "Developing a Product Model Online" in *Developer Guide*.

Figure 2-147 Creating a Modbus product



Adding a Modbus Edge Device

Add a Modbus gateway under the edge node. The procedure is as follows:

Step 1 Log in to the IoTDA console.

Step 2 Choose **IoTEdge** > **Nodes** in the left navigation pane, and click the name of the target edge node to access its details page.

Step 3 Choose **Application Module** and click the **Modules** tab to deploy an edge application.

To ensure that the edge device can go online properly, deploy the edge_access application first. For details, see **Deploying an Application**.

Step 4 Choose **Device**, click the **Edge Device List** tab, click **Add Edge Device**, set the parameters based on **Table 2-42**, and click **OK**.

Figure 2-148 Adding a Modbus edge device

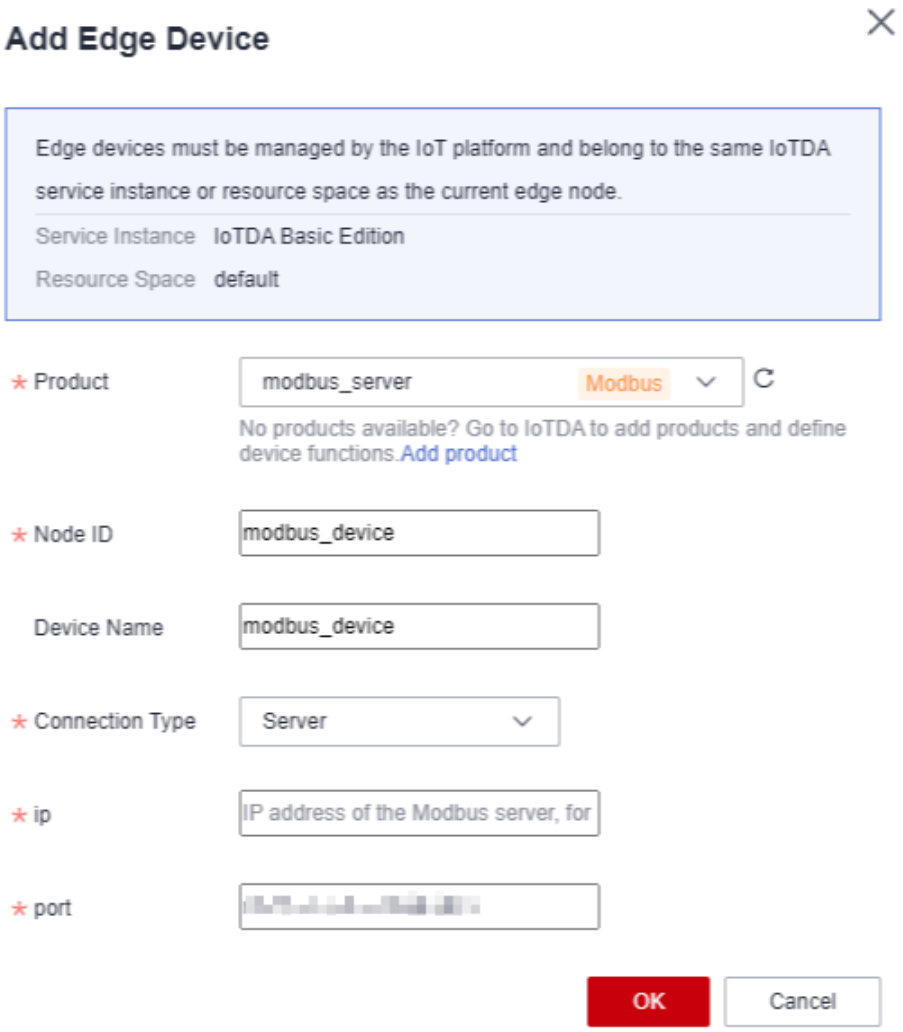


Table 2-42 Parameters for adding a Modbus edge device

Parameter	Description
Product	Select the product name **modbus_server**. **NOTE** • The system provides the preset product model **modbus_server**. Do not delete it. • If the resource space where the edge node is located is not the default resource space, the product name of the Modbus gateway is ***First 24 digits of the resource space ID*+modbus_server**.
Node ID	Specify a unique physical identifier for the device, such as its IMEI or MAC address. This parameter is used by IoTDA to authenticate the device during device access. If no physical device is available, enter a character string consisting of letters and numbers.

Parameter	Description
Device Name	Customize a value.
Connection Type	• If you select **Server**, set the following parameters: – **IP**: IP address of the Modbus server – **Port**: port for connecting to the Modbus server, for example, 502 • If you select **Client**, you must select an authentication type. – None – **Password**: secret details

Step 5 View the device status and manage the device.

You can click a device ID to access the **Device Details** page.

Figure 2-149 Managing an edge device

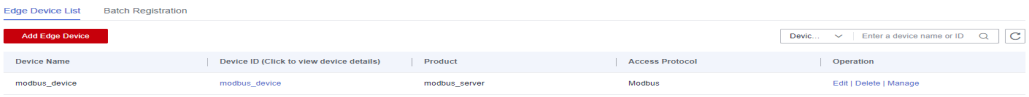


Table 2-43 Operations

Parameter	Description
Edit	Check or modify the configuration of an edge device.
Delete	Delete an edge device.
Manage	View details about the edge device and register a child device. For details, see **Registering a Modbus Child Device**.

----End

Registering a Modbus Child Device

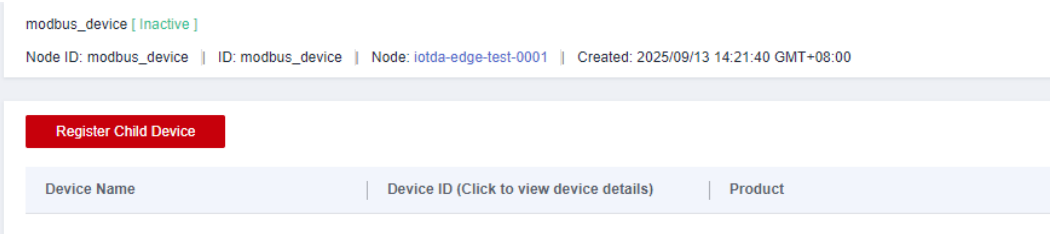
Register a Modbus child device under the Modbus gateway and bind the child device to the specified Modbus gateway. The procedure is as follows:

Step 1 Choose **IoTEdge** > **Nodes** in the left navigation pane, and click the name of the target edge node to access its details page.

Step 2 Choose **Device** and click the **Edge Device List** tab. On the displayed page, click **Manage** on the right of the device.

Step 3 Click **Register Child Device**.

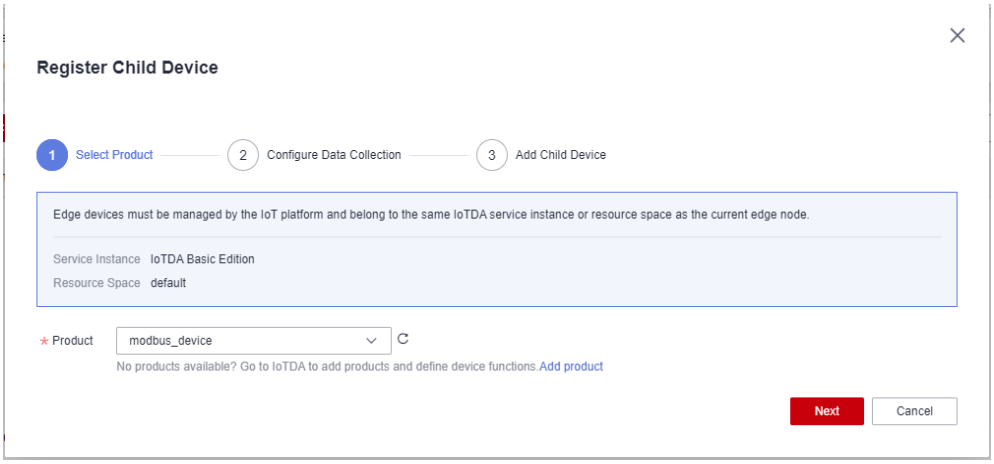
Figure 2-150 Registering a Modbus child device



Step 4 Register a Modbus child device.

1. Select the product created in **Developing a Product Model for a Modbus Device**.

   Figure 2-151 Selecting a product

2. Configure data collection. If you use a new product to register a Modbus child device for the first time, you must configure data collection. You can modify device configuration using batch property delivery. You can configure product properties by following the instructions provided in **Table 2-44**.

Figure 2-152 Configure Data Collection tab page

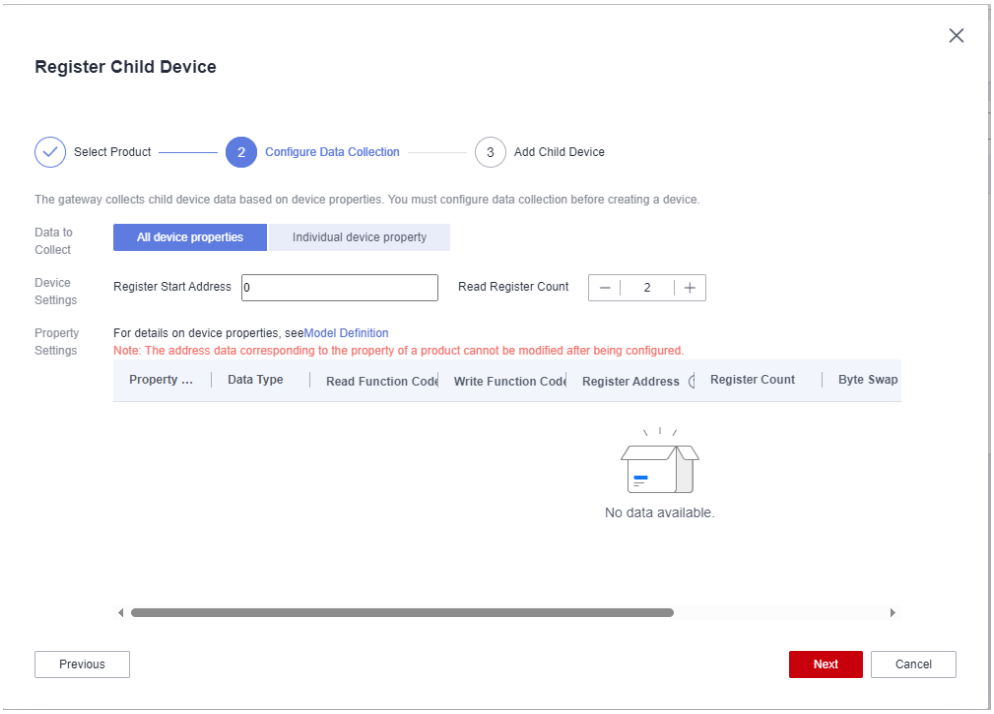


Table 2-44 Modbus child device properties

Parameter	Description
Data to Collect	– **All device properties**: An instruction is used to collect all properties of a device. The collection address range must be supported by the sensor. It is recommended that the collection address range of **All device properties** be the same as that of a single measurement point. – **Individual device property**: An instruction is used to collect one property of a device.
Device Settings	Set this parameter when **Data to Collect** is set to **All device properties**. **Register Start Address**: Customize a value. **Read Register Count**: Customize a value.

Parameter		Description
Property Settings	Read Function Code	Specify the read command provided by the Modbus device. Read function codes are classified into bit access function codes and 16-bit access function codes. The function codes are decimal numbers. Bit access function codes: – 01: read coils – 02: read discrete inputs 16-bit access function codes: – 03: read holding registers – 04: read input register
	Write Function Code	Specify the write command provided by Modbus devices to servers. Write function codes are classified into bit access function codes and 16-bit access function codes. The function codes are decimal numbers. Bit access function codes: – 05: write single coil – 15: write multiple coils 16-bit access function codes: – 06: write single register – 16: write multiple registers **NOTE** When the device property values are stored in the unit of register, it is recommended that **Read Function Code** be **3** and **Write Function Code** be **16**.
	Register Address	Specify the address of the register that stores the properties of the Modbus device. Each register address occupies 16 bits.
	Register Count	Specify the number of registers where the property data is located.
	Byte Swap	Specify whether to swap the upper and lower bits of the data in the register. The default value is **false**. For example, if the property data stored in the register is **0xabcd**, the data obtained by the edge node after byte swap is **0xcdab**.
	Register Swap	Specify whether to swap the register position. The default value is **false**. For example, if the start address of register A is 0001 and that of register B is 0002, after register swap, the start address register A is 0002 and that of register B is 0001.

Parameter		Description
	Scale Factor	The data in the register is multiplied by the scale factor to obtain the required data. For example, if the obtained temperature data is 365 and the scale factor is 0.1, the obtained actual temperature data is 365 x 0.1 = 36.5

3. Add the child device. In the dialog box displayed, set the parameters based on **Table 2-45** and click **OK**.

Figure 2-153 Register Child Device dialog box

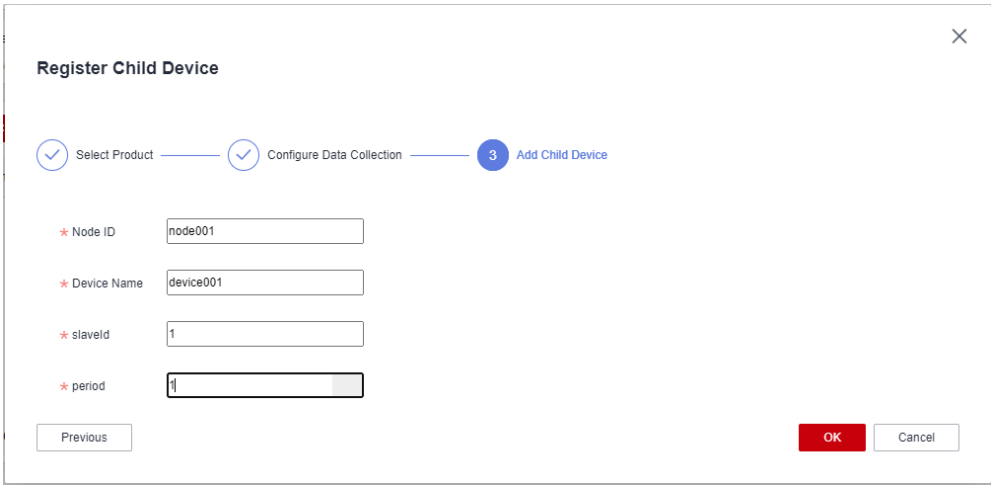


Table 2-45 Parameters for adding a child device

Parameter	Description
Node ID	Specify a unique physical identifier for the device, such as its IMEI or MAC address. This parameter is used by IoTDA to authenticate the device during device access. If no physical device is available, enter a character string consisting of letters and numbers.
Device Name	Customize a value.
slaveId	Specify the ID of the secondary site. Slave IDs identify different Modbus devices in the same channel. The value of this parameter must be the same as the slave number planned for the Modbus device.
period	Specify the data collection interval. Set an interval at which the edge node collects Modbus device data. The unit is second. The minimum value is 1 second. Set this parameter based on the actual data collection period of the Modbus device.

Step 5 Power on the Modbus device to connect it to the edge node. After a data collection period elapses, you can view the collected device data in the device list.

----End

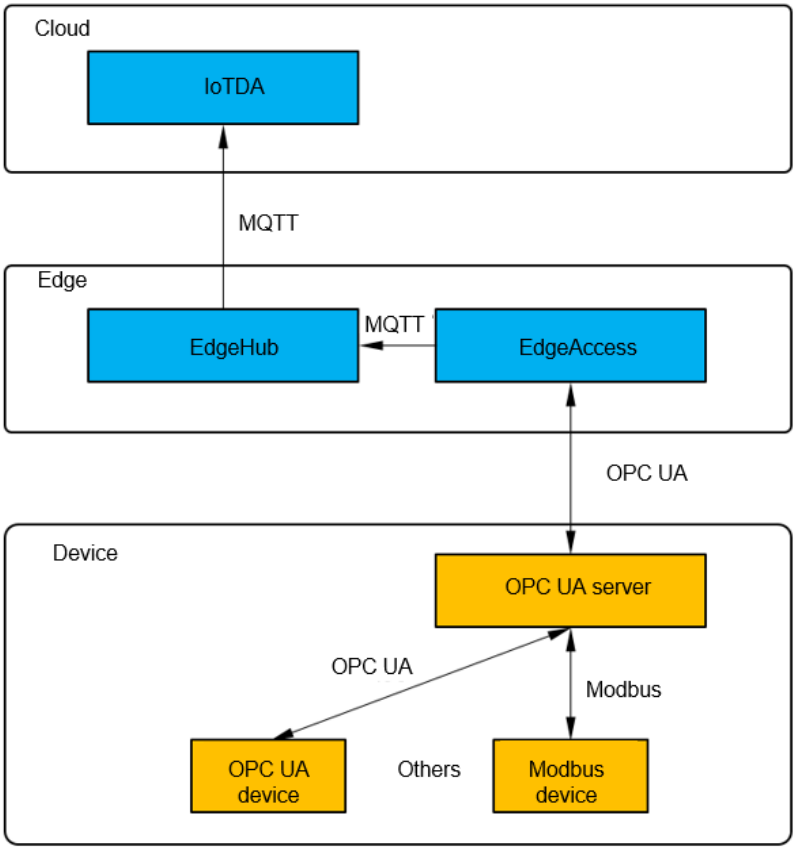
2.11.2.2.2 OPC UA Device Access

Overview

OPC UA is widely used for communication between industrial equipment, thanks to its robust features such as cross-platform, service-oriented architecture, and secure communication.

Devices using OPC UA are not directly connected to IoTEdge nodes.

Figure 2-154 OPC UA network topology



To connect OPC UA devices to an IoTEdge node, a server that supports OPC UA must be deployed on the edge side. Devices can access the server using OPC, OPC UA, or other protocols. The EdgeAccess module deployed on the edge node supports OPC UA access. The edge node functions as the OPC UA client to communicate with the server. It uses the information configured during device registration to browse the corresponding node on the server and perform interaction such as subscription and data writing.

To connect an OPC UA device to an IoTEdge node, you must register an OPC UA gateway with the edge node. Then register the OPC UA child device under the

OPC UA gateway, and configure the corresponding node path and subscription period. The OPC UA server pushes data to the edge node based on the subscription period when the child device data changes. The edge node reports the data to IoTDA using MQTTS based on the product model.

During data collection, the edge node can instruct the gateway to collect binary data from OPC UA devices at a specific interval. After data collection is complete, the edge node normalizes the collected data into JSON format and reports the data to IoTDA using MQTTS.

Connection Procedure

Scenario	Procedure	Description
Device preparation	Prepare an OPC UA gateway, which functions as the server.	Purchase a gateway that supports OPC UA.
	Prepare an OPC UA device.	Apply for a sensor that supports OPC UA.
Operations on IoTDA	Develop a product model for the OPC UA edge device.	Develop a product model for the OPC UA edge device, which is the gateway, on the IoTDA console.
	Add an OPC UA edge device.	Add an OPC UA edge device under the edge node.
	Register an OPC UA child device.	Register an OPC UA child device under the OPC UA edge device.

Developing a Product Model for an OPC UA Edge Device

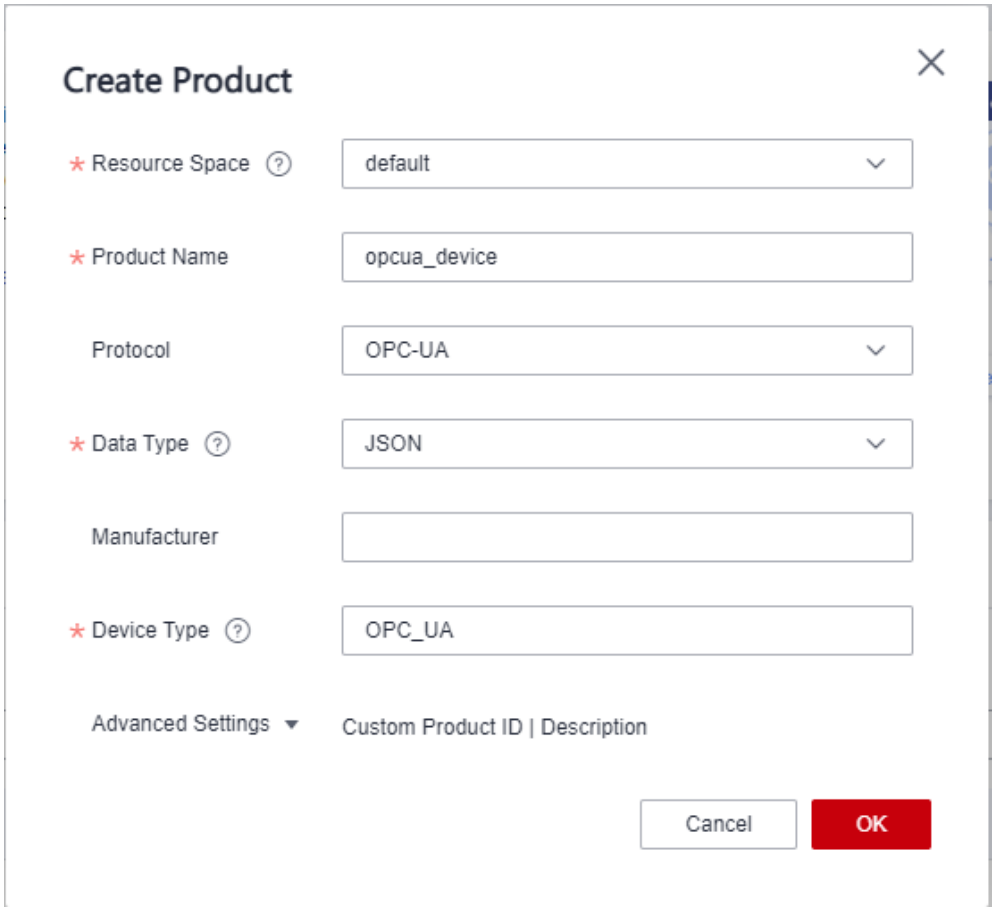
To develop a product model for OPC UA device capabilities, **create a product**, define the product model, and build an abstract model for the device on IoTDA, so that IoTDA can understand the services, properties, and commands supported by the OPC UA device, such as the temperature and battery level. For details, see "Product Development" > "Developing a Product Model" > "Developing a Product Model Online" in *Developer Guide*.

Create a product whose protocol is OPC UA.

CAUTION

- **Protocol** (mandatory): **OPC-UA**
- After creating a product, add services and properties for the product. An empty product cannot be used to create a device. For details, see "Product Development" > "Developing a Product Model" > "Developing a Product Model Online" in *Developer Guide*.

Figure 2-155 Creating an OPC UA product



Adding an OPC UA Edge Device

Add an OPC UA edge device under the edge node.

Step 1 Log in to the IoTDA console.

Step 2 Choose **IoTEdge** > **Nodes** in the left navigation pane, and click the name of the target edge node to access its details page.

Step 3 Choose **Application Module** and click the **Modules** tab to deploy an edge application.

To ensure that the edge device can go online properly, deploy the edge_access application first. For details, see **Deploying an Application**.

Step 4 Choose **Device**, click the **Edge Device List** tab, click **Add Edge Device**, set the parameters based on **Table 2-46**, and click **OK**.

Figure 2-156 Adding an OPC UA edge device

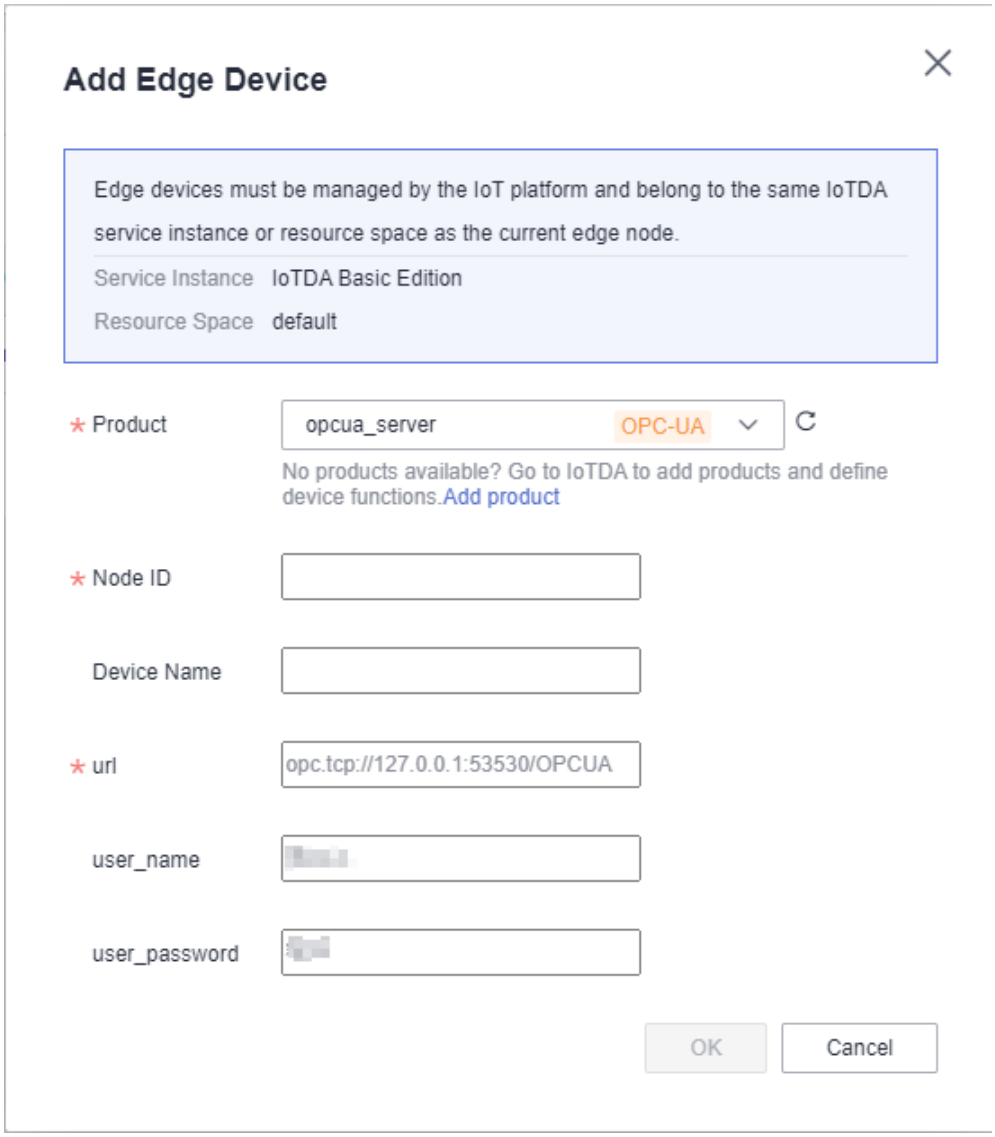


Table 2-46 Parameters for adding an OPC UA edge device

Parameter	Description
Product	Select the product name **opcua_server**. **NOTE** • The system provides the preset product model **opcua_server**. Do not delete it. • If the edge node is not in the default resource space, the product name of the OPC UA edge device is ***First 24 digits of the resource space ID*+opcua_server**.
Node ID	Specify a unique physical identifier for the device, such as its IMEI or MAC address. This parameter is used by IoTDA to authenticate the device during device access. If no physical device is available, enter a character string consisting of letters and numbers.

Parameter	Description
Device Name	Customize a value.
url	Enter the OPC UA access address provided by the OPC UA server, for example, **opc.tcp://192.168.1.1:4840**.
user_name	Enter the username used by the edge node to connect to the OPC UA server. This parameter is optional.
user_password	Enter the password used by the edge node to connect to the OPC UA server. This parameter is optional.

Step 5 View the device status and manage the device. You can also click its device ID to access the IoTDA console for device query and management.

If the edge node is connected to the OPC UA server, the device status is **Online**.

Figure 2-157 Managing an OPC UA edge device

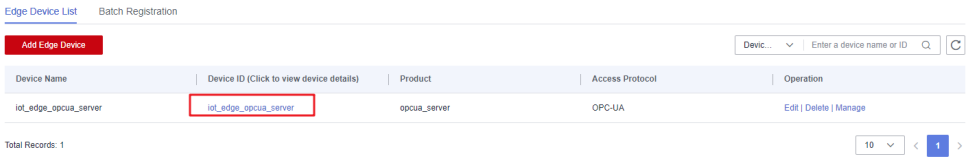


Table 2-47 Operations

Parameter	Description
Edit	Check or modify the configuration of an edge device.
Delete	Delete an edge device. **NOTE** After deleting an edge device on the IoTDA console, log in to the IoTEdge console, access the edge node details page, and delete the device on the **Edge Devices** tab page. Otherwise, you will still be billed for the device by IoTEdge.
Manage	View details about the edge device and register a child device. For details, see **Registering an OPC UA Child Device**.

----End

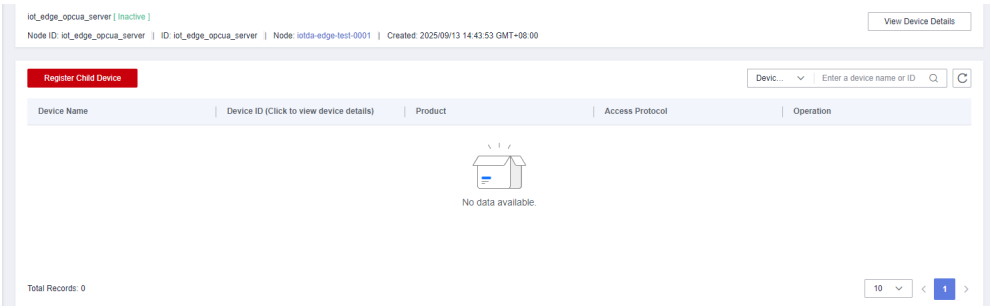
Registering an OPC UA Child Device

Register an OPC UA child device under the OPC UA edge device.

Step 1 Choose **Device** and click the **Edge Device List** tab. On the displayed page, click **Manage** on the right of the device.

Step 2 Click **Register Child Device**.

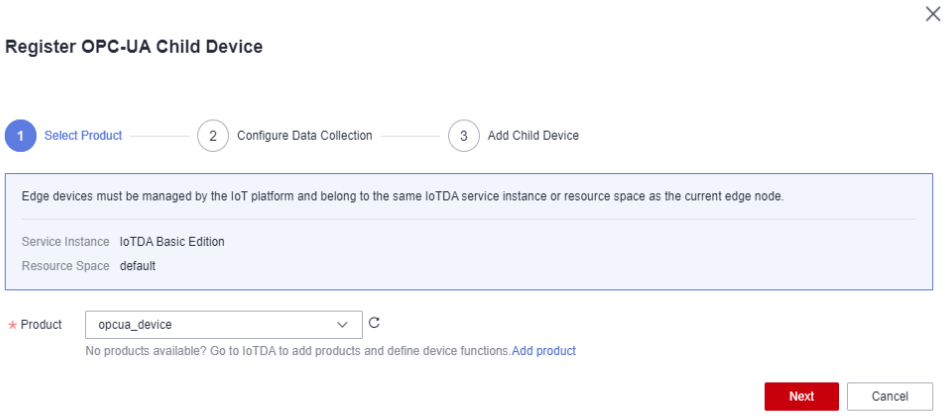
Figure 2-158 Register an OPC UA child device.



Step 3 Register an OPC UA child device.

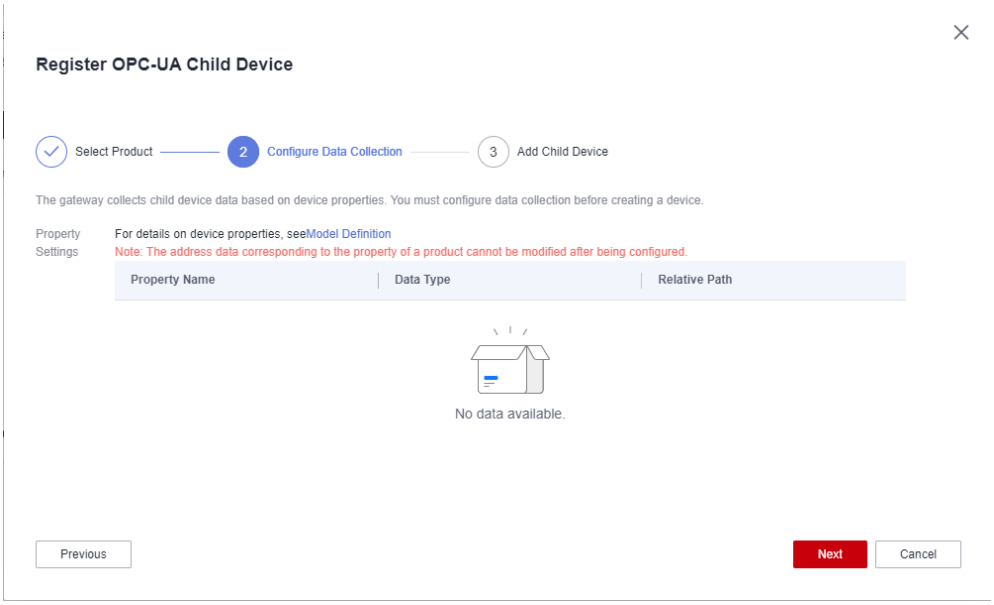
1. Select a product. Select the product created in **Developing a Product Model for an OPC UA Edge Device**.

   Figure 2-159 Selecting a product



2. Configure data collection. If you use a new product to register an OPC UA child device for the first time, configure a relative path for each property of the device. You can modify configuration using batch property delivery. The relative path is the relative position of the device property in the node path of the OPC UA server address space. It is in the array format. Each item in the array is the BrowseName of each layer from the node path to the device property node.

Figure 2-160 Configure Data Collection tab page



3. Add the child device. In the dialog box displayed, set the parameters based on **Table 2-48** and click **OK**.

Figure 2-161 Add Child Device dialog box

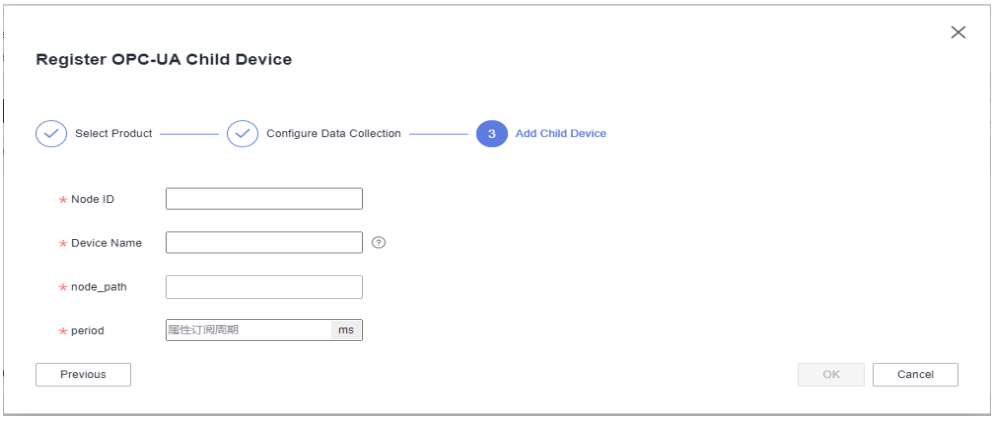


Table 2-48 Parameters for adding a child device

Parameter	Description
Node ID	Specify a unique physical identifier for the device, such as its IMEI or MAC address. This parameter is used by IoTDA to authenticate the device during device access. If no physical device is available, enter a character string consisting of letters and numbers.
Device Name	Customize a value.

Parameter	Description
node_path	Enter the path of a device node, which is the absolute path from the Objects node to the device node in the OPC UA server address space. The device node refers to the node where the absolute paths of property nodes intersect in the address space. The combination of the device node path and the relative path of the property is the absolute path of the device property node in the address space of the OPC UA server.
period	Enter the interval at which the OPC UA server pushes data to the edge node when device data changes, in ms.

Step 4 After the preceding operations are complete, the edge node subscribes to the property node of the OPC UA device from the OPC UA server. If the OPC UA device has been connected to the server, the edge node can successfully subscribe to and read the property data and report the data. In the future, data will be reported based on the subscription period when data changes.

----End

2.11.2.3 Transparent Transmission Gateway

The transparent transmission gateway mode applies to the following scenarios:

1. Devices that support the TCP/IP protocol stack can directly communicate with the IoTDA platform, but are expected to **directly connect to edge nodes** for near-end management.
2. Devices that do not support the TCP/IP protocol stack cannot directly communicate with the platform. They must **connect to edge nodes through gateways** for near-end management.

NOTICE

Table 2-41 describes the capabilities of IoTEdge to transparently transmit IoTDA packets.

Direct Connection Between a Device and an Edge Node

A device is directly connected to an edge node by using the native MQTT protocol or integrating IoT Device SDK or IoT Device SDK Tiny. The edge node transparently transmits data reported by the device to the IoTDA platform and commands delivered by the IoTDA platform to the device.

Perform the following operations to connect a device to an edge node:

Step 1 Log in to the IoTDA console.

Step 2 Develop a product model (also called profile) and codec, and perform self-service tests.

The cloud platform must be able to parse data reported by the device through the edge node.

For details, see "Product Development" in *Developer Guide*.

- A **profile** is a JSON file that describes device capabilities. It defines basic device properties and message formats for data reporting and command delivery. Defining a profile is to build an abstract model of a device on the IoTDA platform. When the device reports data, the IoTDA platform can understand the properties supported by the device based on the defined profile.

Step 3 Perform device-side development for connecting the device to the IoTDA platform.

1. Select an appropriate access mode based on the device. You can select the native MQTT protocol or the SDK integration provided by the platform.

   For details, see "Development on the Device Side" in *Developer Guide*.

   a. If you use the native MQTT protocol for access, see "Using MQTT Demos for Access".

   b. If you use the provided SDK for access, see "Using IoT Device SDKs for Access".

2. Change the device access IP address to the local IP address of the edge node to be connected.

   NOTE

   Enable the device to access the edge node through port 7883 (for MQTT devices) and download the server certificate for authentication. To download the certificate, click **Download CA Certificate** on the **Overview** page of the IoTDA console.

Step 4 Add an edge device.

1. Log in to the IoTDA console.
2. Choose **IoTEdge** > **Nodes** in the left navigation pane, and click the name of the target edge node to access its details page.
3. (Optional) Choose **Application Module** and click the **Modules** tab to deploy an edge application.

   To ensure that the device can be properly online, deploy the application in **Step 3** first. In the direct connection scenario, you do not need to deploy other applications. For details, see **Deploying an Application**.
4. Choose **Device**, click the **Edge Device List** tab, click **Add Edge Device**, set the parameters based on **Table 2-49**, and click **OK**.

Figure 2-162 Adding an edge device

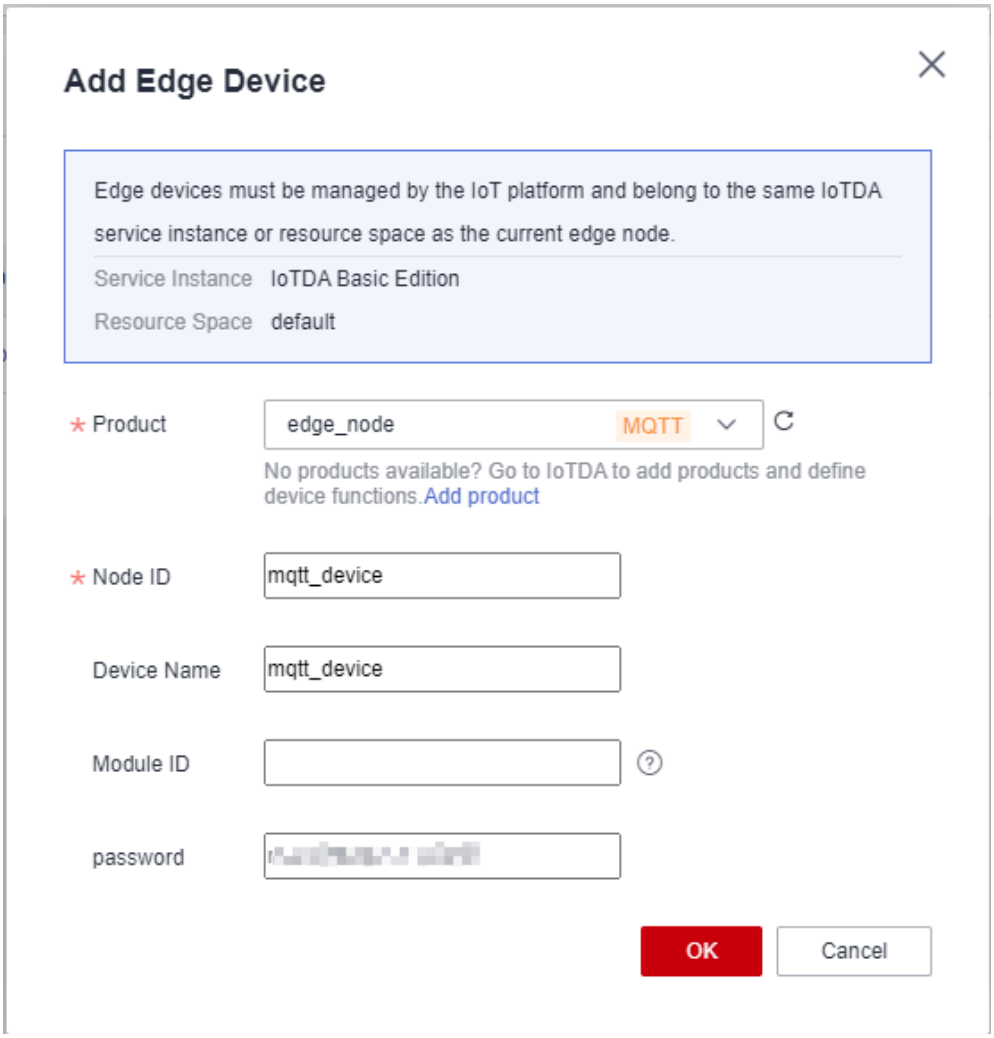


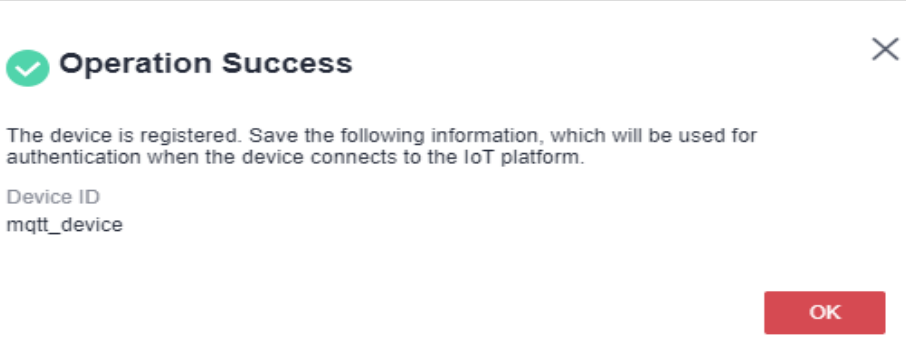
Table 2-49 Parameters for adding an edge device

Parameter	Description
Product	Select a product. You can select a product only after it is defined on the IoTDA console. If no product is available, **create a product** first.
Node ID	Specify a unique physical identifier for the device, such as its IMEI or MAC address. This parameter is used by the IoTDA platform to authenticate the device during device access. For MQTT devices, the device ID (corresponding to the node ID) and secret generated after the registration are used for device access.
Device Name	Customize a value.

Parameter	Description
Module ID	Leave it blank. **NOTE** By default, you do not need to enter a module ID. If the application is integrated with the module SDK, you need to configure a forwarding rule and enter the module ID of the third-party application so that data can be reported to the platform.
password	Customize a device secret.

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

Figure 2-163 Device ID and secret



NOTE

Use the MQTT simulator to connect to IoTEdge. For details, see "Using MQTT Demos for Access" > "Debugging Using an MQTT Simulator" in *Developer Guide*.

6. View the device status in the device list and manage the device.

   You can click a device ID to access the **Device Details** page.

Table 2-50 Operations

Operation	Description
Edit	Check or modify the configuration of an edge device.
Delete	Delete an edge device.
Manage	View details about the edge device and register a child device.

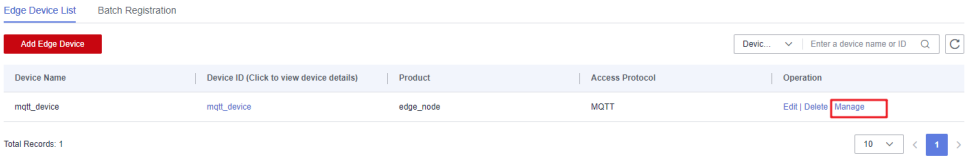
Step 5 Connect the device to an edge node.

Step 6 Verify that data reported by a device can be viewed on the device details page.

1. Power on the child device to report data to the IoTDA platform.
2. Log in to the IoTDA console.

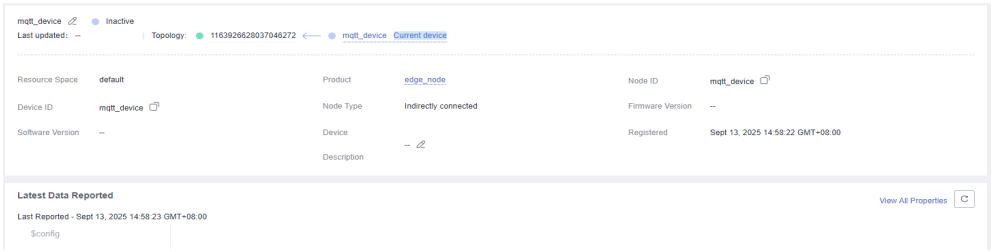
3. Choose **IoTEdge** > **Nodes** in the left navigation pane, and click the name of the target edge node to access its details page.
4. Choose **Device**, click the **Edge Device List** tab, and click **Manage** to view the device status. If the status is **Online**, the device has been connected to the IoTDA platform.

Figure 2-164 Managing a child device



5. Return to the edge device list page. Click the device ID. On the details page, view the latest reported data. If the data can be properly parsed and displayed, the data reporting is successful.

Figure 2-165 Viewing the latest reported data



----End

Connection Between a Device and an Edge Node Through a Gateway

Devices use simple near-field communication protocols, such as ZigBee, Z-Wave, and Bluetooth, or other non-IP wired transmission protocols, such as serial ports and parallel ports, to access a gateway. The gateway connects to an edge node by integrating the IoT Device SDK. The edge node transparently transmits data from the gateway to the IoTDA platform and commands from the IoTDA platform to the gateway.

Perform the following operations to connect a device to an edge node:

Step 1 Log in to the IoTDA console.

Step 2 Create a gateway, develop a product model (also called profile) and codec, and perform self-service tests.

For details, see "Product Development" in *Developer Guide*.

The IoTDA platform must be able to parse data reported by the gateway and device through the edge node.

- A **profile** is a JSON file that describes device capabilities. It defines basic device properties and message formats for data reporting and command delivery. Defining a profile is to build an abstract model of a device on the IoTDA platform. When the device reports data, the IoTDA platform can

understand the properties supported by the device based on the defined profile.

Step 3 Perform device-side development for connecting the gateway to the IoTDA platform.

1. Integrate the IoT Device SDK into the gateway to access the IoTDA platform. Currently, IoT Device SDKs using the C and Java programming languages are available. During actual development, select an appropriate SDK for integration based on the programming language and platform used during development. For details, see "Development on the Device Side" > "Using IoT Device SDKs for Access" in *Developer Guide*.
2. Change the gateway access IP address to the local IP address of the edge node to be connected.

   NOTE

   Enable the MQTT gateway to connect to the edge node via port 7883.

Step 4 Add a gateway. In the IoTDA platform, the gateway is registered as a device. For details, see **Step 4**.

Step 5 Connect the gateway to the edge node.

Step 6 Connect the child device to the gateway.

Step 7 Verify that data reported by a device can be viewed on the device details page.

1. Power on the child device to report data to the IoTDA platform.
2. Log in to the IoTDA console.
3. Choose **IoTEdge** > **Nodes** in the left navigation pane, and click the name of the target edge node to access its details page.
4. Choose **Device**, click the **Edge Device List** tab, and click the device ID to access the device details page.
5. Click the **Child Devices** tab, and view the device status in the device list. If the status is **Online**, the device has been connected to the IoTDA platform.

   NOTE

   The status of a child device indicates whether the child device is connected to the gateway, and the gateway reports the status to the platform for status updates. If the gateway does not report the status of a child device, the child device status is not updated on the platform. For example, after a child device connects to the platform through a gateway, the child device status is displayed as online. If the gateway is disconnected from the platform, the gateway can no longer report the child device status and the platform will consider the child device online.
6. Click **View** to view the details, historical data, and operation records of the child device.

----End

2.11.3 Application Management

2.11.3.1 Overview

IoTEdge expands application management capabilities to the edge by deploying preset or custom applications on edge nodes. **Table 2-51** describes the preset applications in the system.

Table 2-51 Preset applications

Application Name	Application Type	Description
$edge_hub	Mandatory	It is the processing center on the edge node and is responsible for device and communication management.
$edge_agent	Mandatory	It manages edge applications on edge nodes, including deployment, upgrade, and running monitoring.
$edge_access	Optional	It extends the protocol access capability of edge nodes. Currently, Modbus and OPC-UA protocols are supported.
$edge_apigw	Optional	API gateway of the edge node, which provides request routing as well as forward and reverse proxy for edge applications. It must be used together with route management.
$edge_omagent	Optional	It performs remote monitoring and O&M on edge nodes. It provides log reporting, remote SSH connection, data reporting, and file upload/download.
$edge_push	Optional	It provides the external push capability.
$edgetepa	Optional	It is used for traffic incident detection at the edge.
$ot_dc_db	Optional	It is used for edge data collection.
$edge_keepalive	Optional	It is used for active/standby switchover.
$edge_rule	Optional	It is used to compute and process edge device data.
$industry_dc_bsi	Optional	It is used for IT data collection.

2.11.3.2 Adding a Service Application

Add a custom edge application.

Procedure

Step 1 Log in to the IoTDA console.

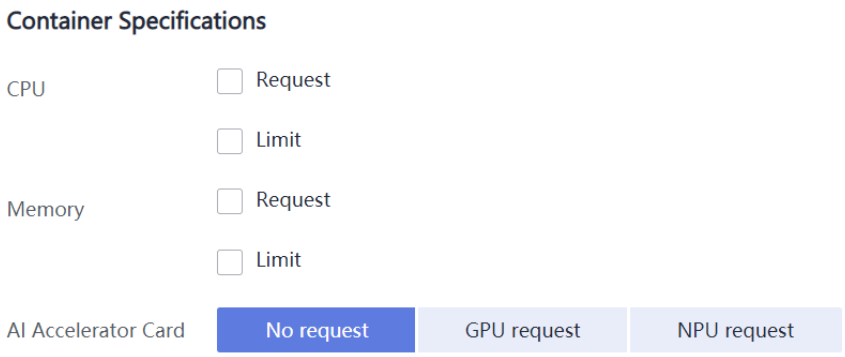
Step 2 Choose **IoTEdge** > **Applications** in the left navigation pane. On the page displayed, click **Service Application**, and click **Add Application**.

Step 3 After the application is added, go to the application version configuration page.

Step 4 On the **Container Settings** page, perform the following steps and click **Next**.

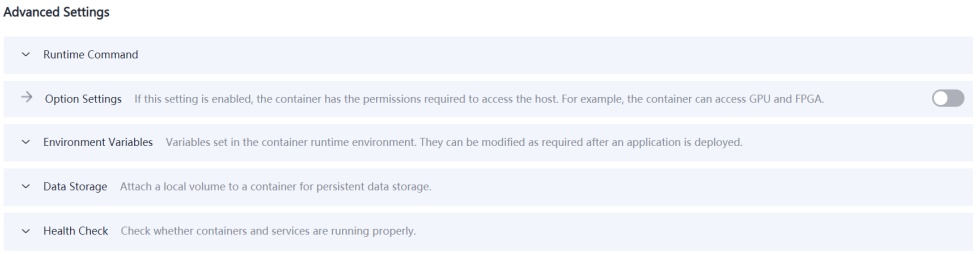
1. Enter the image path or installation package address.
2. Select container specifications as required.

Figure 2-166 Container specifications



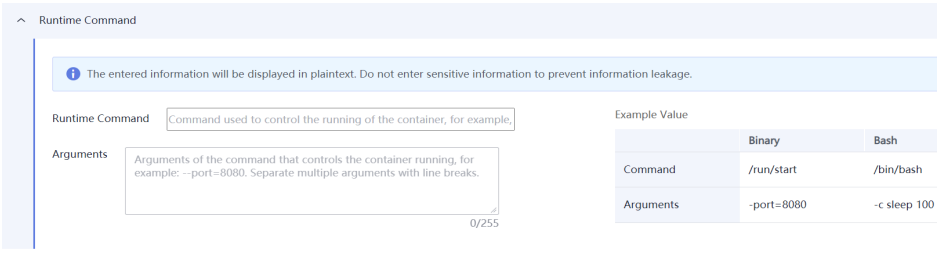
3. Configure advanced settings.

Figure 2-167 Advanced settings



a. **Runtime Command** (which can be configured only in container-based deployment mode)

Figure 2-168 Runtime command



- **Runtime Command**: Enter an executable command, for example, **/run/start**.
  - If there are multiple commands, separate them with spaces, for example, **ls ps**.
  - If a command contains a space, enclose the command in quotation marks (""), for example, **"sleep 100"**.

    NOTE

    If there are many commands, you are advised to run **/bin/sh** or other **shell** commands and use other commands as parameters.

- **Arguments**: Enter an argument that controls the container runtime command, for example, **-port=8080**.

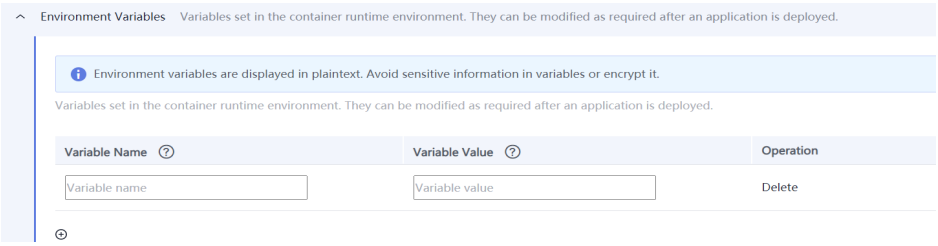
  If there are multiple arguments, separate them with line breaks.

b. **Option Settings**

If this setting is enabled ( ), the container has the permissions required to access devices (such as GPUs and FPGAs) on the host.

c. **Environment Variables**

Figure 2-169 Environment variables



An environment variable is a variable whose value can affect the way a running container will behave. You can modify environment variables even after applications are deployed.

Click and enter a variable name and value.

NOTE

The platform does not encrypt the environment variables you entered.

If the environment variables you attempt to configure contain sensitive information, encrypt them before entering them and decrypt them through applications.

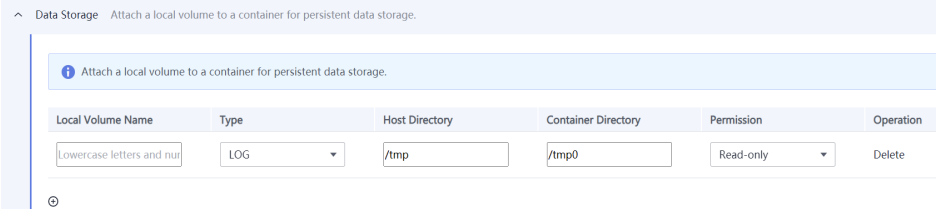
The platform does not provide any encryption and decryption tools. To configure cypher text, use other encryption and decryption tools.

d. **Data Storage**

A volume is a storage volume used for container running.

A volume mounts a directory of the host into the container. Host directory can be used for persistent storage. After an application is deleted, the data in hostPath still exists in the local disk directory of the edge node. If the application is re-created later, existing data can still be read after the directory is mounted.

Figure 2-170 Data storage



NOTE

- The container path must not be a system directory, such as **/** and **/var/run**. Otherwise, an exception occurs. You are advised to mount the container to an empty directory. If the directory is not empty, ensure that the directory does not contain any files that affect container startup. Otherwise, the files will be replaced, making it impossible for the container to be properly started. Consequently, the application creation will fail.
- If the container is mounted into a high-risk directory, you are advised to use an account with minimum permissions to start the container. Otherwise, high-risk files on the host machine may be damaged.
- If the local volume type is set to **LOG**, **DB**, or **CONFIG**, the prefix **/var/IoTEdge/{log|db|config}** is automatically added to the host directory. To mount the container directory to a directory on the host, set the local volume type to **Other**.
- If the disk space of the mounted directory is full, the node becomes abnormal and cannot be used. In this case, clear the disk space in a timely manner.
- The system configures the **config** volume by default. The default mount position is **/config** for the host machine and **/opt/config** for the container.

e. **Health Check**

Health check regularly checks the status of containers or workloads. There is a liveness probe and a readiness probe.

- **Liveness Probe**: The system checks if a container is still alive, and restarts the container if the probe fails. Currently, the system probes a container by HTTP request or command and determines whether the container is alive based on the response from the container.
- **Readiness Probe**: The system determines whether a container is ready. If it is not ready, the system does not forward requests to it.

Figure 2-171 Health check

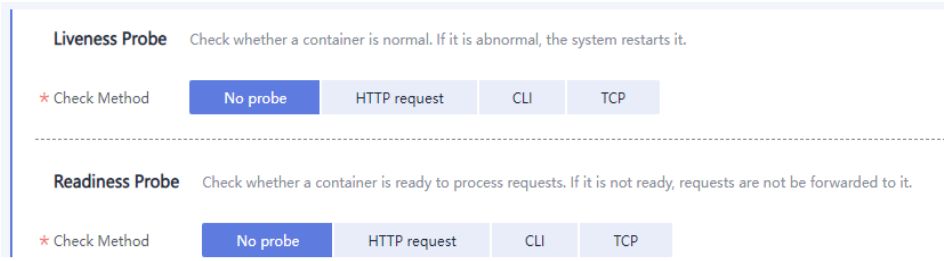


Table 2-52 Check methods

Parameter	Description
HTTP request	IoTEdge periodically initiates an HTTP GET request to a container. If HTTP code 2xx or 3xx is received, the container is healthy. IoTEdge will send an HTTP GET request to **http://{***Instance IP address***}/healthz:8080** 10 seconds after the container starts. If no response is received within 2 seconds, the health check fails. If status code 2xx or 3xx is received, the container is healthy. **NOTE** You do not need to specify the host address. By default, the instance's IP address is used (requests are sent to the container) unless you have special requirements.
CLI check	The probe runs commands in the container and checks the command output. If the command output is 0, the container is healthy. IoTEdge will run **cat /tmp/healthy** 10 seconds after the container starts. If no response is received within 2 seconds, the health check fails. If the command output is 0, the container is healthy.
TCP check	It checks whether the TCP port can be accessed. If it can be accessed, the container is normal.

Figure 2-172 HTTP request

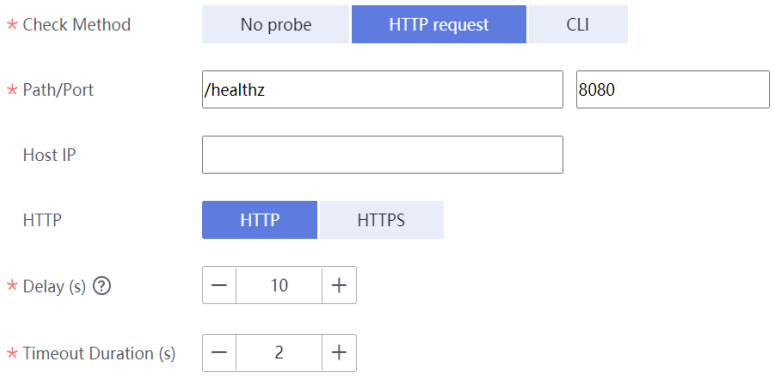


Figure 2-173 CLI check

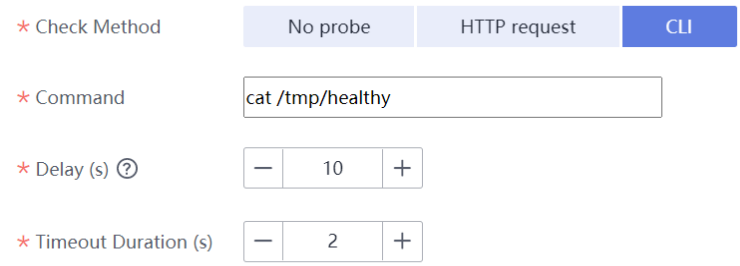
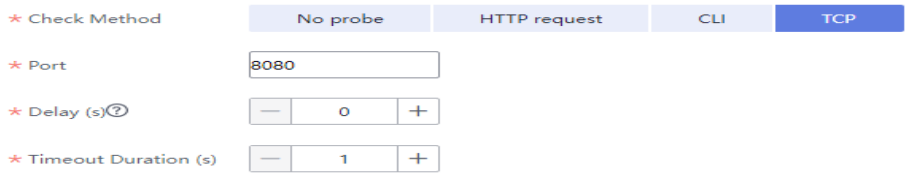
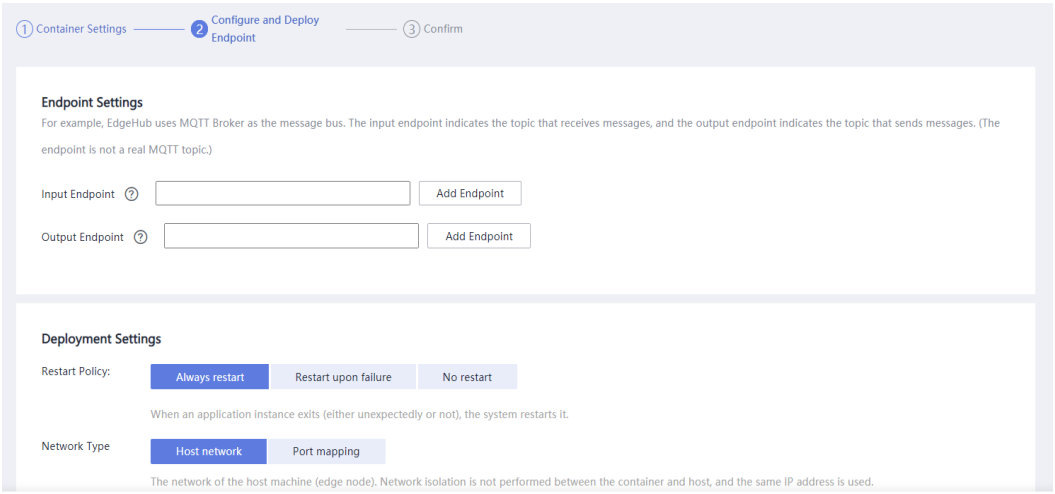


Figure 2-174 TCP check



Step 5 Configure endpoint and deployment settings.

Figure 2-175 Configuring endpoint and deployment settings



1. Configure endpoints.

   EdgeHub uses MQTT as the message bus to communicate with other modules. MQTT functions as the broker to transfer data between EdgeHub and other modules.

   - **Input Endpoint**: suffix of the MQTT topic subscribed by a module. Before sending data to the module, EdgeHub constructs an MQTT topic based on the input endpoint configured for the module.
   - **Output Endpoint**: suffix of the MQTT topic used when data is sent from the module to EdgeHub.

   NOTE

   Only applications that are configured with input and output endpoints can use data forwarding.

2. Deploy the application.

Table 2-53 Deployment parameters

Parameter	Description
Restart Policy	Set this parameter based on service requirements. – **Always restart**: The system restarts an application after it exits normally or unexpectedly. – **Restart upon failure**: The system restarts an application only if it exits unexpectedly. – **No restart**: The system does not restart an application instance regardless of whether it exits (either unexpectedly or not).
Network Type	Containers can be accessed through a host network or using port mapping. – **Host network** The network of the host (edge node) is used. To be specific, the container and the host use the same IP address, and network isolation is not required between them. – **Port mapping** The container uses an independent virtual network. Configure the mapping between container ports and host ports to enable external communications. After the mapping is configured, traffic destined for the host port is directed to the mapping container port. For example, if container port 80 is mapped to host port 8080, the traffic destined for host port 8080 will be directed to container port 80.

Step 6 On the **Confirm** page, set basic information based on **Table 2-54** and click **Next**.

Figure 2-176 Confirm page

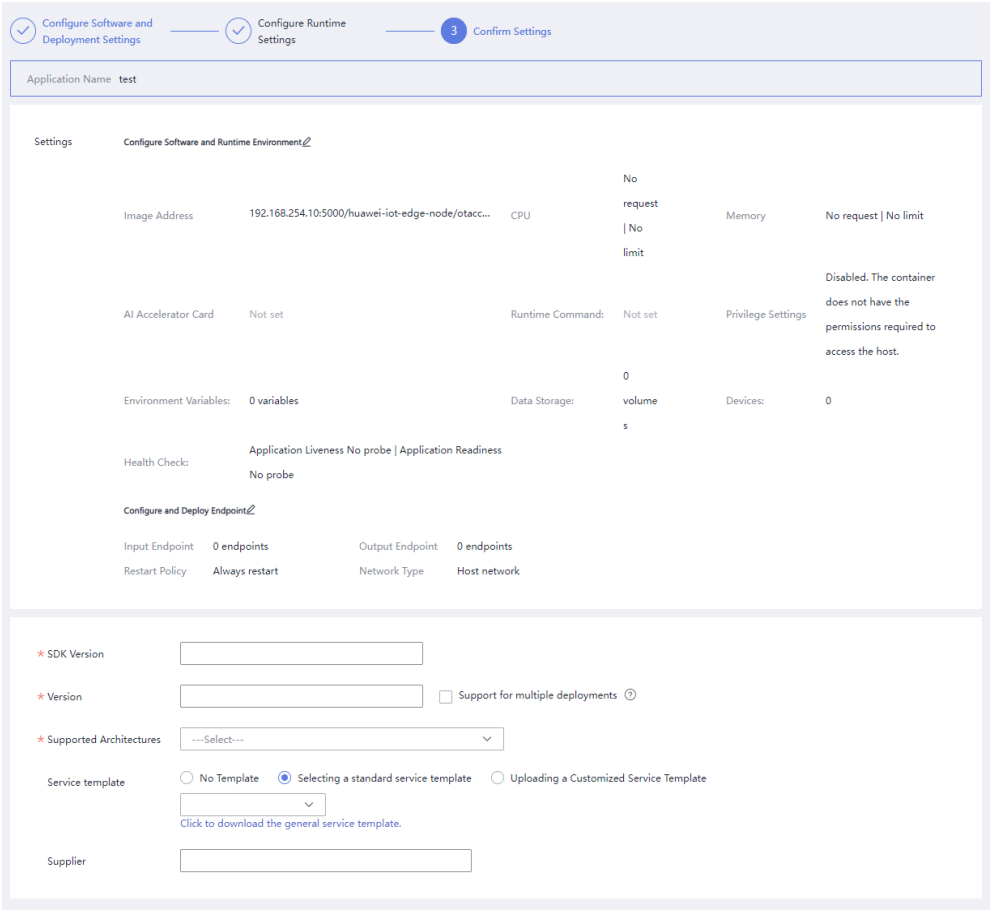


Table 2-54 Container configuration parameters

Parameter	Description
SDK Version	Set the SDK version based on the site requirement.
Version	Customize your edge application version.
Supported Architectures	Select the architecture supported by the edge application, which can be x86_64, Arm64, or both. **NOTE** Currently, Arm32 nodes cannot be deployed.

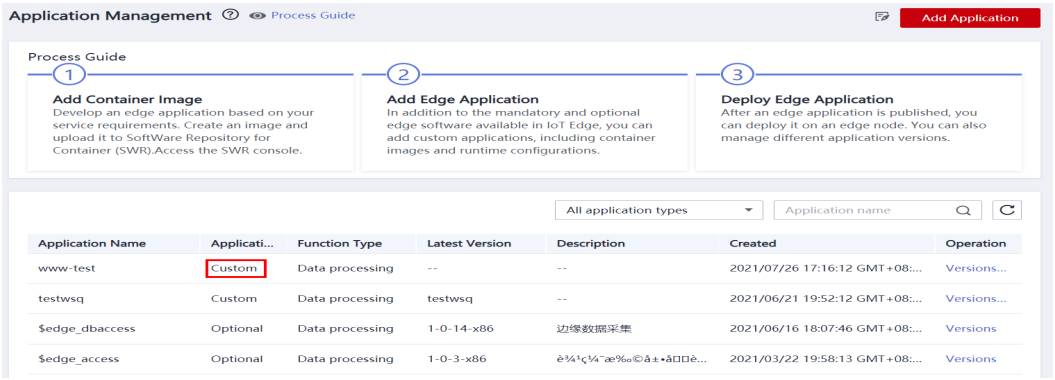
Step 7 Click **OK**. If you want to publish the application version at the same time, select **Publish Now**.

Figure 2-177 Application created

Step 8 Click **Back to Applications**.

You can see that the application type is **Custom**, which is different from the preset applications in the system.

Figure 2-178 Application list



----End

2.11.3.3 Adding a Driver Application

Add a custom edge application.

Procedure

Step 1 Log in to the IoTDA console.

Step 2 Choose **IoTEdge** > **Applications** in the left navigation pane. On the page displayed, click **Driver Applications**, and click **Add Application**.

Table 2-55 Adding a driver application

Parameter	Description
Driver ID	(Mandatory) Customize the value, which must contain at least four characters.
Driver Name	(Mandatory) Customize the value, which must contain at least four characters.

Parameter	Description
Protocol	Protocol type for application integration. If no protocol is used, select **Other**.
Function Type	Select the data collection option.
Application Description	(Optional) Describe the functions and usage of the application.

Step 3 After the application is added, go to the application version configuration page.

Step 4 Configure software and deployment settings.

1. Select a deployment mode and enter basic information by referring to **Table 2-56**.

Figure 2-179 Software and deployment settings

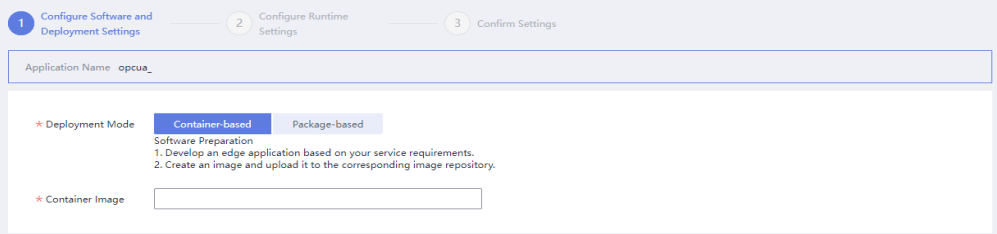
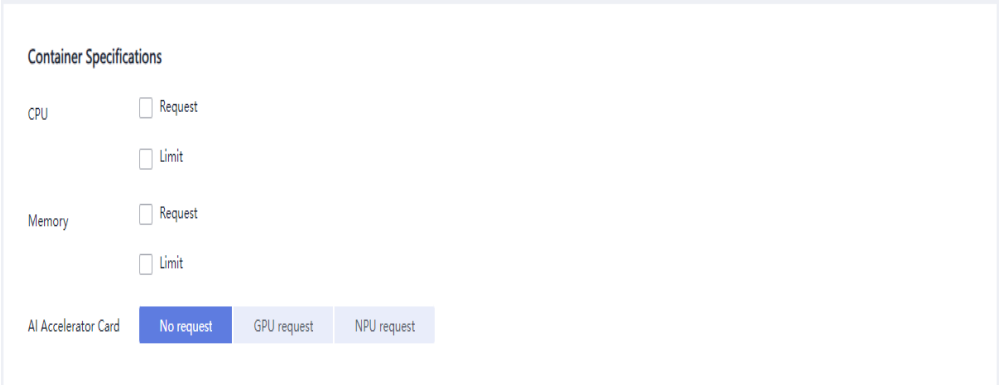


Table 2-56 Deployment parameters

Parameter	Description
Deployment Mode	**Container-based**: The edge application runs as a Docker container on edge nodes. Obtain the containerized application from the corresponding image repository. **Package-based**: The edge application runs as a process on edge nodes. Before adding such an application, ensure that the installation package has been uploaded to Object Storage Service (OBS).
Container Image	If container-based deployment is used, set this parameter to the image repository address.
Installation Package Address	If **Deployment Mode** is set to **Package-based**, enter the installation package address in OBS.

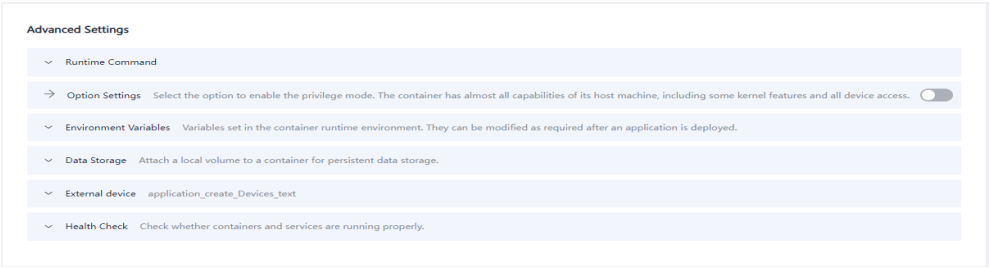
2. Select container specifications as required.

Figure 2-180 Container specifications configuration



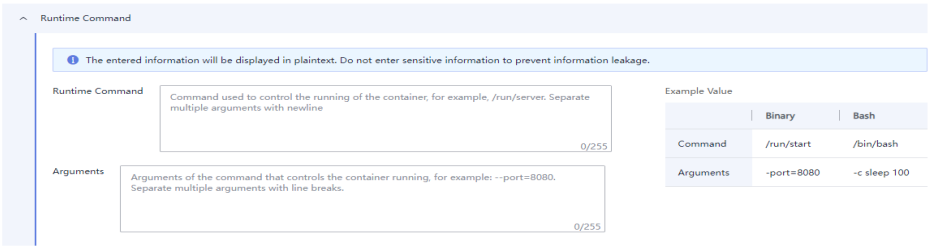
3. Configure advanced settings.

Figure 2-181 Advanced settings



a. **Runtime Command** (which can be configured only in container-based deployment mode)

Figure 2-182 Runtime command



- **Runtime Command**: Enter an executable command, for example, **/run/start**.
  - If there are multiple commands, separate them with spaces, for example, **ls ps**.
  - If a command contains a space, enclose the command in quotation marks (""), for example, **"sleep 100"**.

    NOTE

    If there are many commands, you are advised to run **/bin/sh** or other **shell** commands and use other commands as parameters.

- **Arguments**: Enter an argument that controls the container runtime command, for example, **-port=8080**.

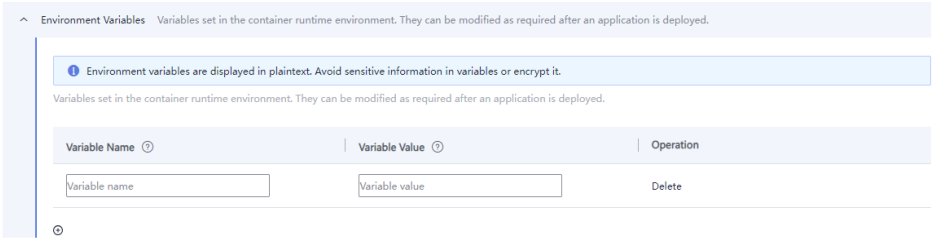
If there are multiple arguments, separate them with line breaks.

b. **Option Settings**

If this setting is enabled (), the container has the permissions required to access devices (such as GPUs and FPGAs) on the host.

c. **Environment Variables**

Figure 2-183 Environment variables



An environment variable is a variable whose value can affect the way a running container will behave. You can modify environment variables even after applications are deployed.

Click  and enter a variable name and value.

NOTE

IoTEdge does not encrypt the environment variables you entered.

If the environment variables you attempt to configure contain sensitive information, encrypt them before entering them and decrypt them through applications.

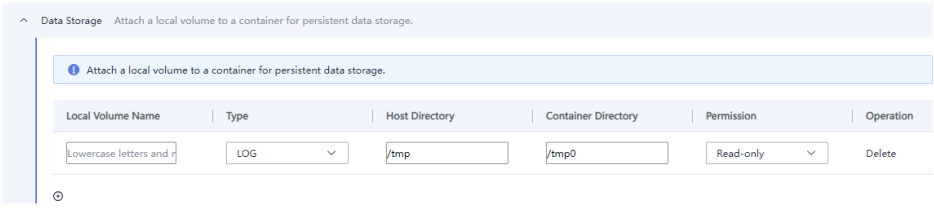
IoTEdge does not provide any encryption and decryption tools. To configure cypher text, use other encryption and decryption tools.

d. **Data Storage**

A volume is a storage volume used for container running.

A volume mounts a directory of the host into the container. Host directory can be used for persistent storage. After an application is deleted, the data in hostPath still exists in the local disk directory of the edge node. If the application is re-created later, existing data can still be read after the directory is mounted.

Figure 2-184 Data storage



NOTE

- The container path must not be a system directory, such as **/** and **/var/run**. Otherwise, an exception occurs. You are advised to mount the container to an empty directory. If the directory is not empty, ensure that the directory does not contain any files that affect container startup. Otherwise, the files will be replaced, making it impossible for the container to be properly started. Consequently, the application creation will fail.
- If the container is mounted into a high-risk directory, you are advised to use an account with minimum permissions to start the container. Otherwise, high-risk files on the host machine may be damaged.
- If the local volume type is set to **LOG**, **DB**, or **CONFIG**, the prefix **/var/IoTEdge/{log|db|config}** is automatically added to the host directory. To mount the container directory to a directory on the host, set the local volume type to **Other**.
- If the disk space of the mounted directory is full, the node becomes abnormal and cannot be used. In this case, clear the disk space in a timely manner.

e. **Health Check**

Health check regularly checks the status of containers or workloads. There is a liveness probe and a readiness probe.

- **Liveness Probe**: The system checks if a container is still alive, and restarts the container if the probe fails. Currently, the system probes a container by HTTP request or command and determines whether the container is alive based on the response from the container.
- **Readiness Probe**: The system determines whether a container is ready. If it is not ready, the system does not forward requests to it.

Figure 2-185 Health check

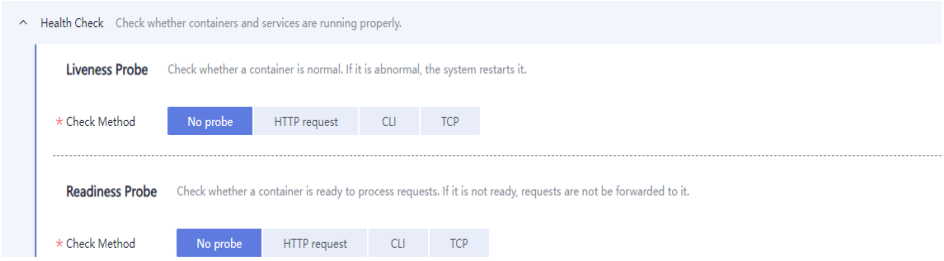


Table 2-57 Check methods

Parameter	Description
HTTP request	IoTEdge periodically initiates an HTTP GET request to a container. If HTTP code 2xx or 3xx is received, the container is healthy. IoTEdge will send an HTTP GET request to **http://***{Instance IP address}***/health:8080** 10 seconds after the container starts. If no response is received within 2 seconds, the health check fails. If status code 2xx or 3xx is received, the container is healthy. **NOTE** You do not need to specify the host address. By default, the instance's IP address is used (requests are sent to the container) unless you have special requirements.
CLI check	The probe runs commands in the container and checks the command output. If the command output is 0, the container is healthy. IoTEdge will run **cat /tmp/healthy** 10 seconds after the container starts. If no response is received within 2 seconds, the health check fails. If the command output is 0, the container is healthy.
TCP check	It checks whether the TCP port can be accessed. If it can be accessed, the container is normal.

Figure 2-186 HTTP request

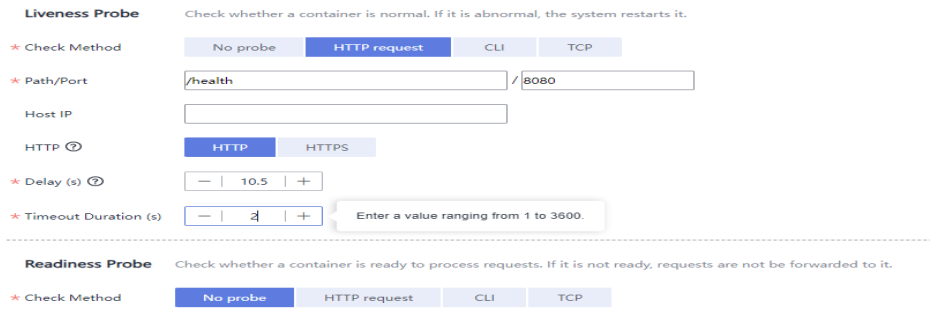


Figure 2-187 CLI check

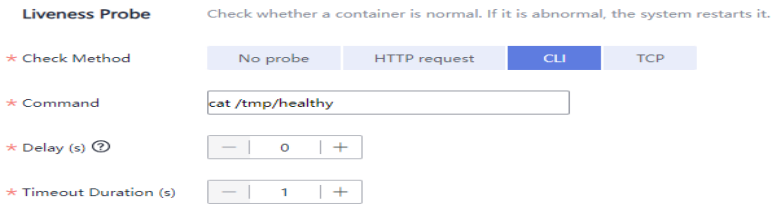
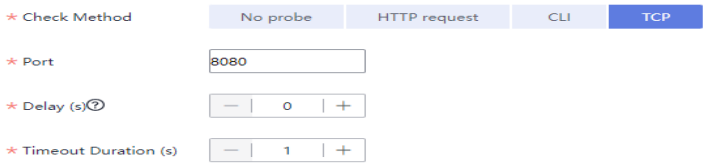
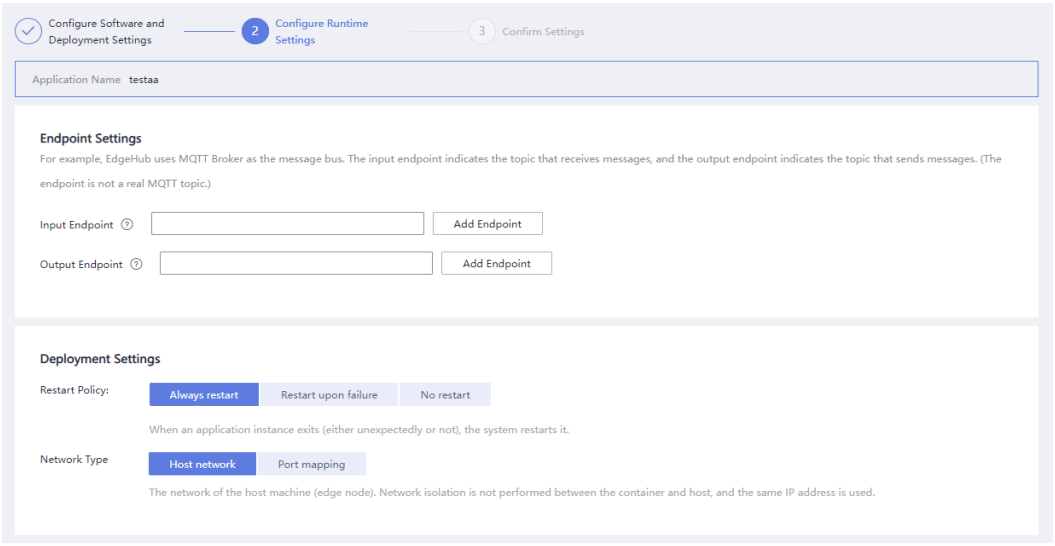


Figure 2-188 TCP check



Step 5 Configure runtime settings.

Figure 2-189 Configuring endpoint and deployment settings



1. Configure endpoints.

   EdgeHub uses MQTT as the message bus to communicate with other modules. MQTT functions as the broker to transfer data between EdgeHub and other modules.

   - **Input Endpoint**: suffix of the MQTT topic subscribed by a module. Before sending data to the module, EdgeHub constructs an MQTT topic based on the input endpoint configured for the module.
   - **Output Endpoint**: suffix of the MQTT topic used when data is sent from the module to EdgeHub.

   NOTE

   Only applications that are configured with input and output endpoints can use data forwarding.

2. Deploy the application.

Table 2-58 Deployment parameters

Parameter	Description
Restart Policy	Set this parameter based on service requirements. – **Always restart**: The system restarts an application after it exits normally or unexpectedly. – **Restart upon failure**: The system restarts an application only if it exits unexpectedly. – **No restart**: The system does not restart an application instance after it exits normally or unexpectedly.
Network Type	Containers can be accessed through a host network or using port mapping. – **Host network** The network of the host (edge node) is used. To be specific, the container and the host use the same IP address, and network isolation is not required between them. – **Port mapping** The container uses an independent virtual network. Configure the mapping between container ports and host ports to enable external communications. After the mapping is configured, traffic destined for the host port is directed to the mapping container port. For example, if container port 80 is mapped to host port 8080, the traffic destined for host port 8080 will be directed to container port 80.

Step 6 Confirm the settings. Enter the basic information by referring to **Table 2-59**.

Figure 2-190 Confirming configuration

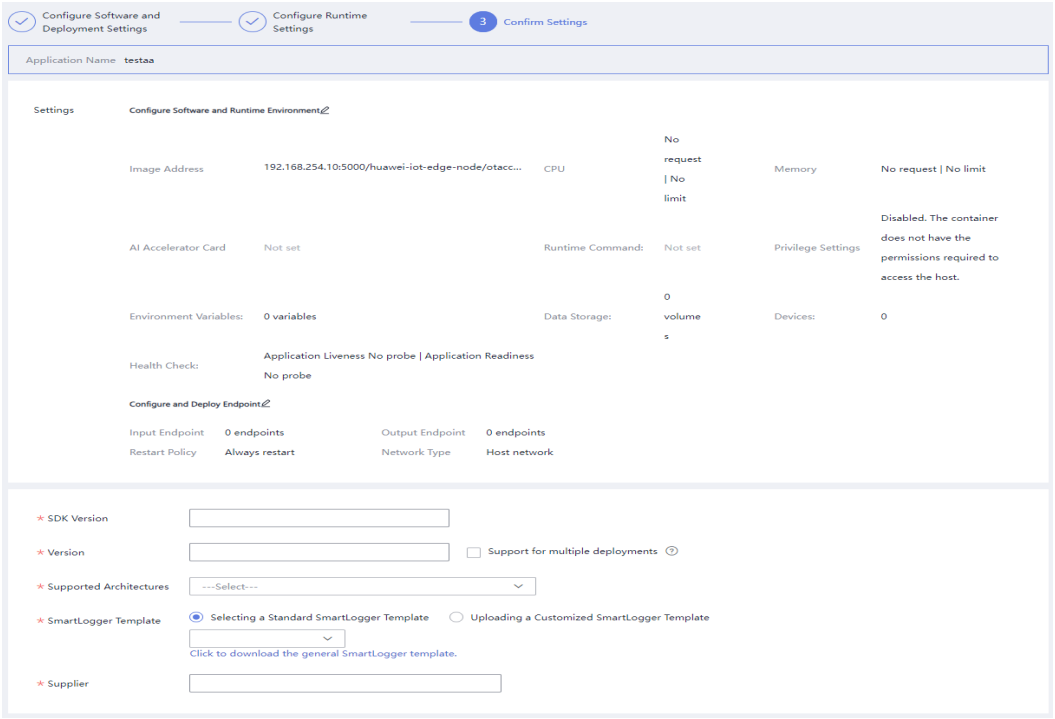
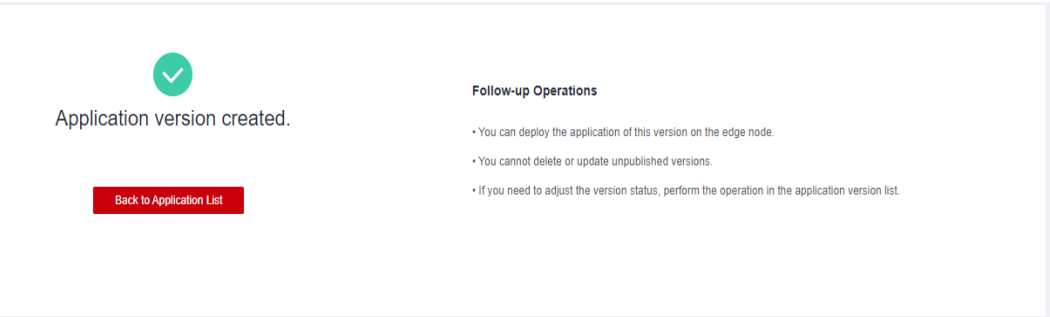


Table 2-59 Application version settings

Parameter	Description
Application Name	Custom edge application name.
SDK Version	Version number of the integrated edge SDK.
Version	Customize your edge application version. You can choose whether to publish it immediately.
Supported Architectures	Select the architecture supported by the edge application. The options are **x86_64**, **arm32**, and **arm64**. You can select multiple options.
Support for multiple deployments	Whether the application version supports the deployment of multiple instances on an edge node.
Data Collection Template	Template used by driver application OT data collection.

Step 7 Click **OK** to complete the application creation. Alternatively, click **Publish Now** to complete the application creation and release the new version.

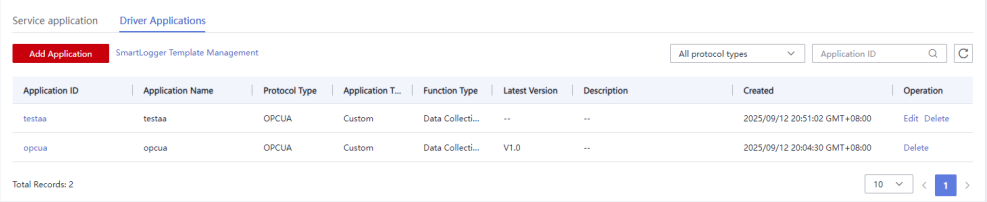
Figure 2-191 Application created



Step 8 Click **Back to Applications**.

You can see that the application type is **Custom**, which is different from the preset applications in the system.

Figure 2-192 Application list



----End

2.11.3.4 Adding a Version

You can create multiple application versions to facilitate application management.

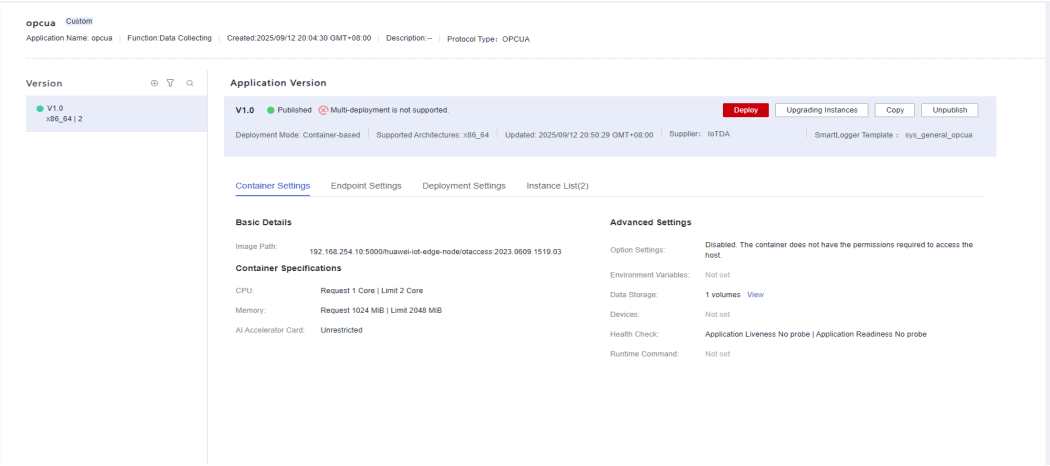
Procedure

Step 1 Log in to the IoTDA console.

Step 2 Choose **IoTEdge** > **Applications** in the left navigation pane.

Step 3 Select the name of the application for which you want to add a version. The application details page is displayed.

Figure 2-193 Application details



Step 4 Click **Add Version** to add an edge application version.

Set the parameters based on **Adding a Service Application**.

----End

2.11.3.5 Deploying an Application

After an edge node is installed, you can deploy edge applications. Pay attention to the following points during the deployment:

CAUTION

- In the standard edition, sys_edge_hub and sys_edge_agent are deployed by default.
- Only published application versions can be deployed.
- Each application can be deployed only once on an edge node. After the application is deleted, it can be deployed again.
- An application can be deployed only when the architecture supported by the application is the same as that supported by the edge node.
- If an application requires an AI accelerator card, the deployment will fail if the edge node does not have an AI accelerator card.
- Application modules can be upgraded, either to a later version or an earlier version. Currently, only the Agent application ($edge_agent) that fails to be upgraded can be rolled back to the source version.

Procedure

Step 1 Log in to the IoTDA console.

Step 2 Choose **IoTEdge** > **Nodes** in the left navigation pane.

Step 3 Click the target edge node name to access its details page.

Step 4 Choose **Application Module**, click the **Modules** tab, and click **Deploy Application**.

Step 5 Configure the parameters as prompted and click **OK**.

Figure 2-194 Deploying an application

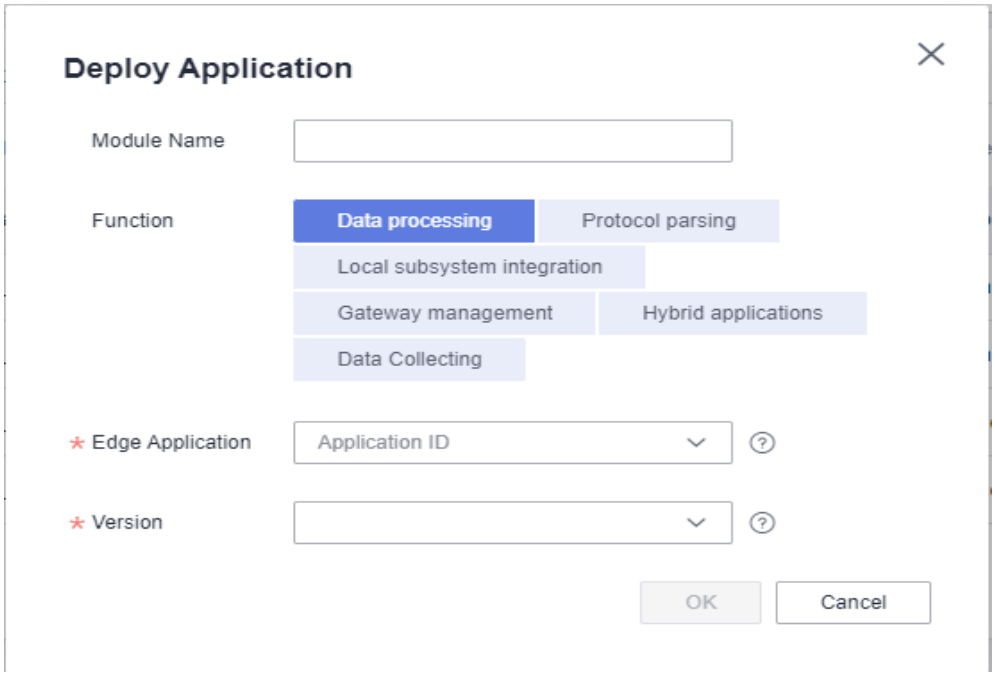
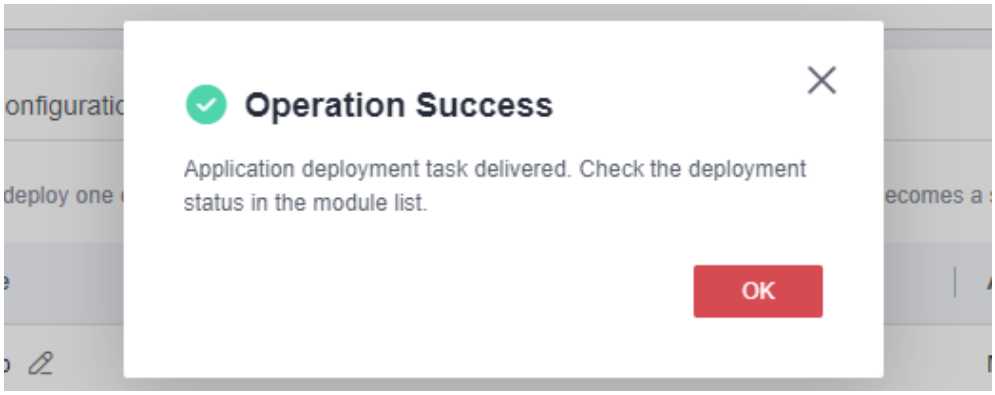


Table 2-60 Parameters for deploying an application

Parameter	Description
Edge Application	Enter the name of the edge application to deploy.
Version	Select a version for the application.
Module	Customize a value.

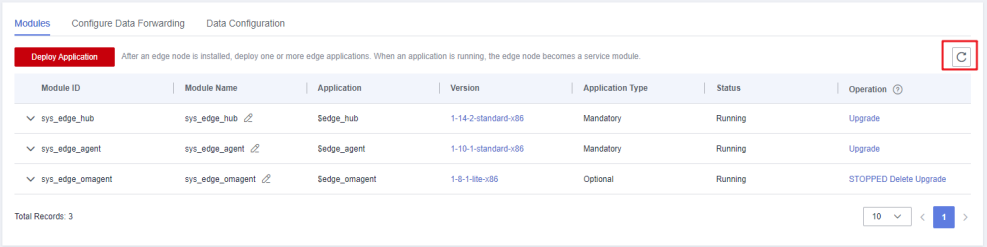
Step 6 In the **Operation Success** dialog box displayed, click **OK** to return to the edge application deployment list.

Figure 2-195 Confirmation



Step 7 Click **Refresh**. If the status of the application instance changes from **Deploying** to **Running**, the deployment is successful.

Figure 2-196 Deployment list



----End

Adding a Port Mapping

You can add a port mapping during or after application deployment only when the application version supports the multiple deployment mode and the network type is port mapping.

For details about the runtime configuration, see the step "Configure endpoint and deployment settings" in "Adding a Service Application".

Figure 2-197 Deploying an Application

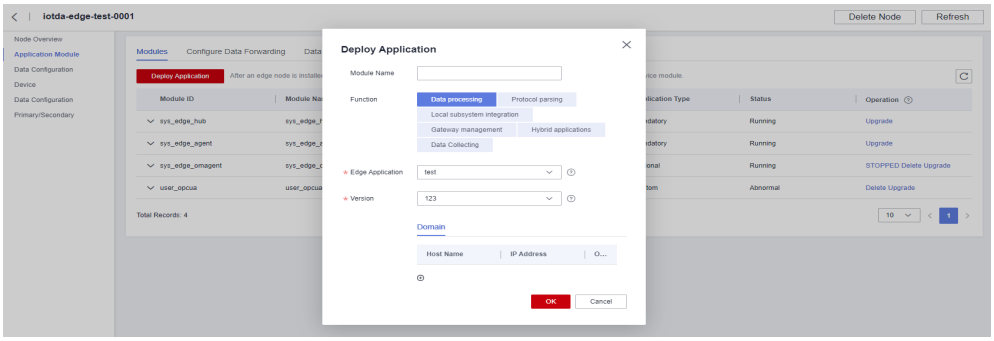
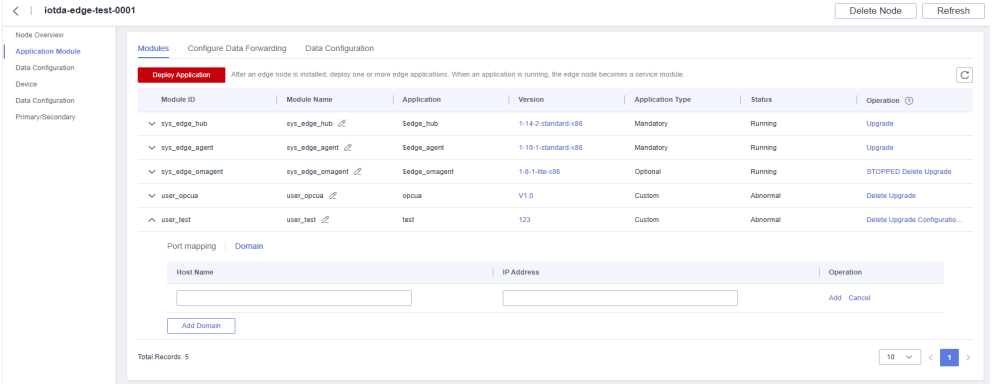


Figure 2-198 Adding a Port Mapping



CAUTION

- By default, sys_edge_hub and sys_edge_agent are deployed on an IoTEdge standard node, and sys_edge_hub is deployed on an IoTEdge advanced node.
- By default, $edge_omagent is deployed on the standard and lite nodes. You can determine whether to automatically deploy it during node registration.
- Only published application versions can be deployed.
- If multi-module deployment is configured when you add an application, the application can be deployed on the same node for multiple times.
- An application can be deployed only when the architecture supported by the application is the same as that supported by the edge node.
- If an application requires an AI accelerator card, the deployment will fail if the edge node does not have an AI accelerator card.
- Application modules can be upgraded, either to a later version or an earlier version. Currently, only the Agent application that fails to be upgraded can be rolled back to the source version.

2.11.3.6 Managing an Application

You can manage IoTEdge applications. After an application is added, you can edit, publish, copy, and delete the application version.

Viewing Application Details

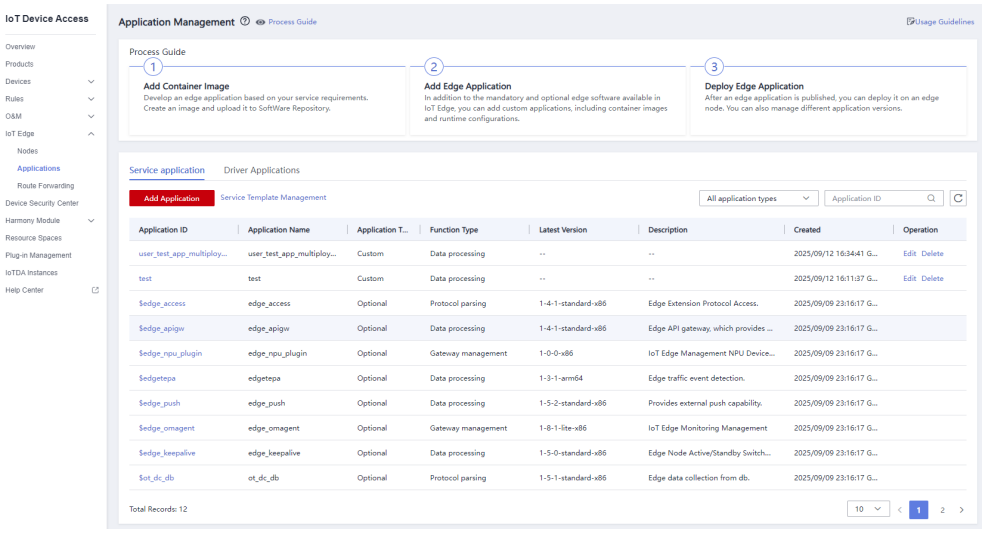
All edge applications are displayed in the edge application list.

You can view the application type, deployment mode, function type, latest version, description, creation time, and operation.

NOTICE

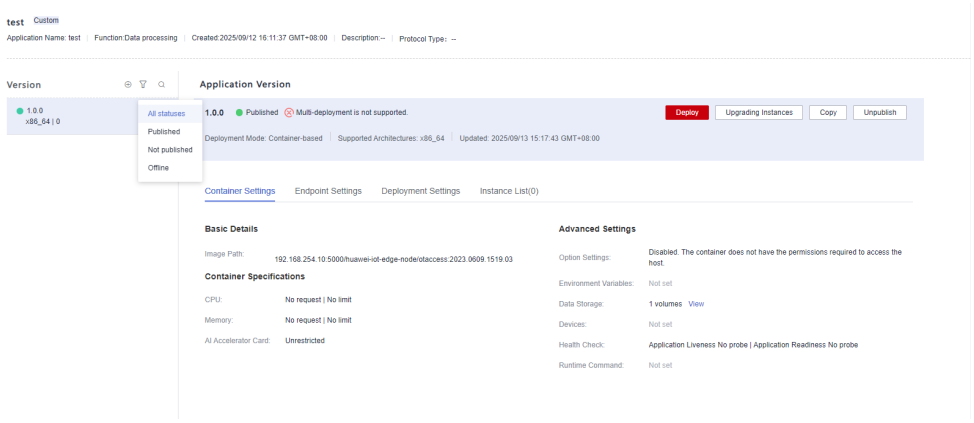
An application that contains a version cannot be deleted. To delete an application, delete the versions of the application first.

Figure 2-199 Viewing the application management list



In the application list, click an application name to go to the application details page. You can view the latest version details of the application.

Figure 2-200 Version details



On the application details page, click the application name on the left to view the version details.

NOTE

Published: Only published versions can be used to deploy instances. Published versions cannot be edited or deleted.

Not published: You can edit and publish a version in the **Not published** state.

Unpublished: A version in the **Unpublished** state can be copied and deleted.

Deploy: You can deploy the application on online nodes in batches.

Upgrade Instances: You can upgrade the application on online nodes in batches.

Table 2-61 Parameter description

Parameter	Description
Publish	After you click **Publish**, the version status changes from **Not published** to **Published**, and then you can deploy instances. A published application cannot be deleted.
Copy	You can copy the configuration of an existing version to quickly create a version.
Unpublish	You can unpublish a version in the **Published** state.
Delete	You can delete a version that is no longer required. **NOTE** You can delete a version in the **Not published** state or a version in the **Unpublished** state that has not been deployed. If the unpublished version has been deployed on a node, the version cannot be deleted.
Deploy	Select an edge node to deploy an instance. After the deployment is complete, you can view the instance status in the instance list on the version details page.

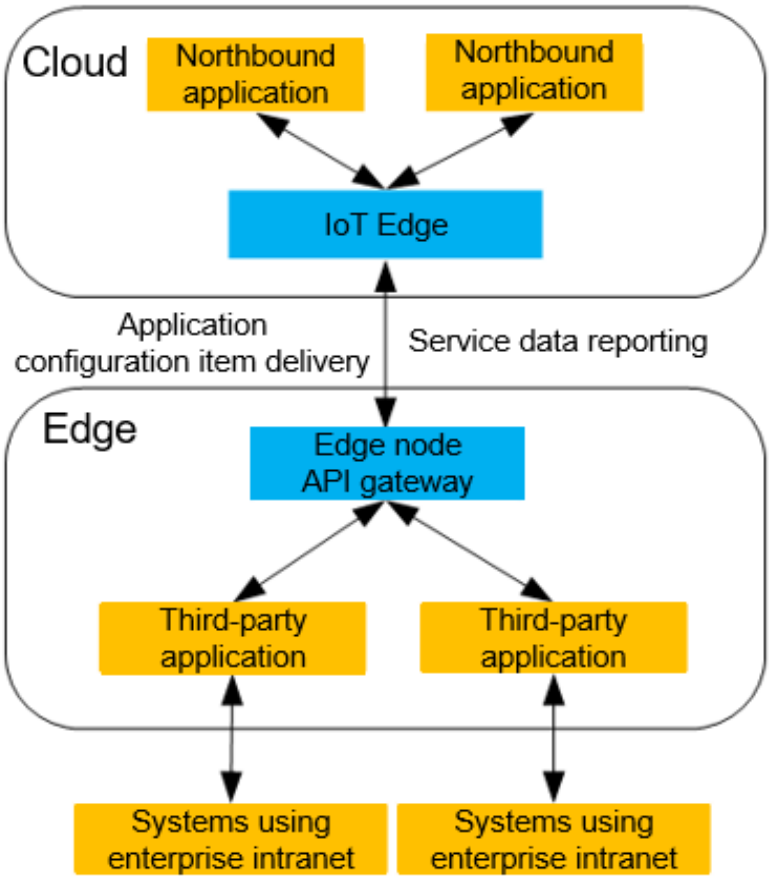
2.11.4 IT Subsystem Integration

2.11.4.1 Overview

IoTEdge is integrated with the industrial cloud subsystem to enable you to call southbound third-party application APIs (including delivering configuration items to third-party applications). It also enables you to report service data of each industrial subsystem collected by southbound third-party applications to northbound applications through the API gateway deployed on the edge node.

Route management is used to manage northbound applications (also called data receiving endpoints). You can add, modify, and delete northbound applications, and grant permissions of northbound applications to one or more edge nodes, which can then report service data.

Figure 2-201 Route management process



2.11.4.2 Route Configuration

Route configuration allows you to manage northbound applications, including adding, deleting, modifying, and authorizing northbound applications.

Procedure

Step 1 Log in to the IoTDA console.

Step 2 In the navigation pane, choose **IoTEdge** > **Routes**. Click **Add Data Receiving Endpoint** in the upper right corner and specify the parameters based on the following table.

NOTE

You can add up to 10 data receiving endpoints.

Figure 2-202 Adding a data receiving endpoint

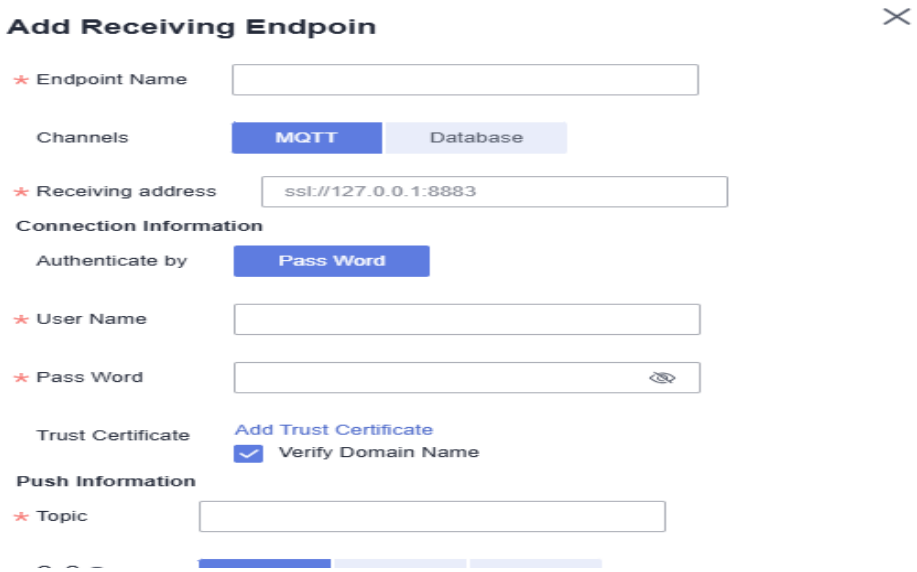


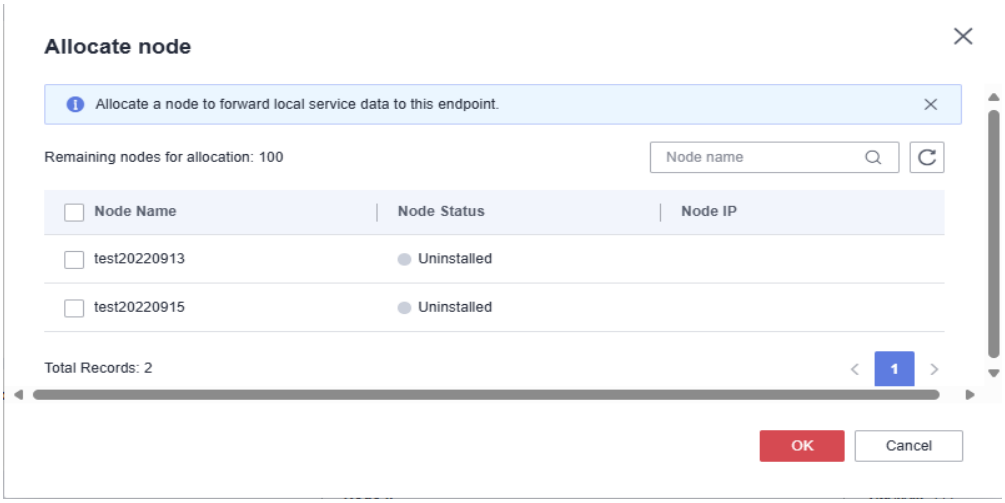
Table 2-62 Parameters for adding a data receiving endpoint

Parameter	Description
Endpoint Name	Enter the name of an endpoint.
Endpoint ID	Enter the ID of the endpoint, which uniquely identifies a northbound application. If you select **Identity authentication required for external access**, you must enter the secret key (SK). Then the edge API gateway will calculate the signature and send the signature to the external endpoint during data forwarding.
Address for Receiving Data	Enter the subdomain name of the API group. The system automatically creates a temporary subdomain name, which is used only for development and testing. This subdomain name can be accessed 1000 times per day. You must bind an independent domain name to the API group to develop services.
Access Mode	Edge nodes call APIs to forward service data. Select the cloud service that the APIs depend on. • If **ROMA** is selected, you must create an integrated application on ROMA, create and publish APIs, and enter the key and secret of the integrated application.
Description	Provide a description for the endpoint.

Step 3 Send local service data to the endpoint.

1. Click the name of the created endpoint. On the displayed page, click **Allocate Node**, select the node to be allocated, and click **OK**.

Figure 2-203 Allocating nodes



2. A message is displayed in the upper right corner, indicating that the allocation is successful.

----End

2.11.4.3 Module Configuration

After a route is configured, you must configure and manage modules. Module configuration allows you to manage the configuration items of third-party applications deployed on edge nodes. You can easily add or delete module configuration items.

Prerequisites

The module configuration function can be used only when the edge node to configure meets the following conditions:

- The edge_apigw application is deployed on the edge node. This application is used to obtain configuration items.
- A third-party application (custom application) whose function is local subsystem integration is deployed on the edge node.

Procedure

Step 1 Log in to the IoTDA console.

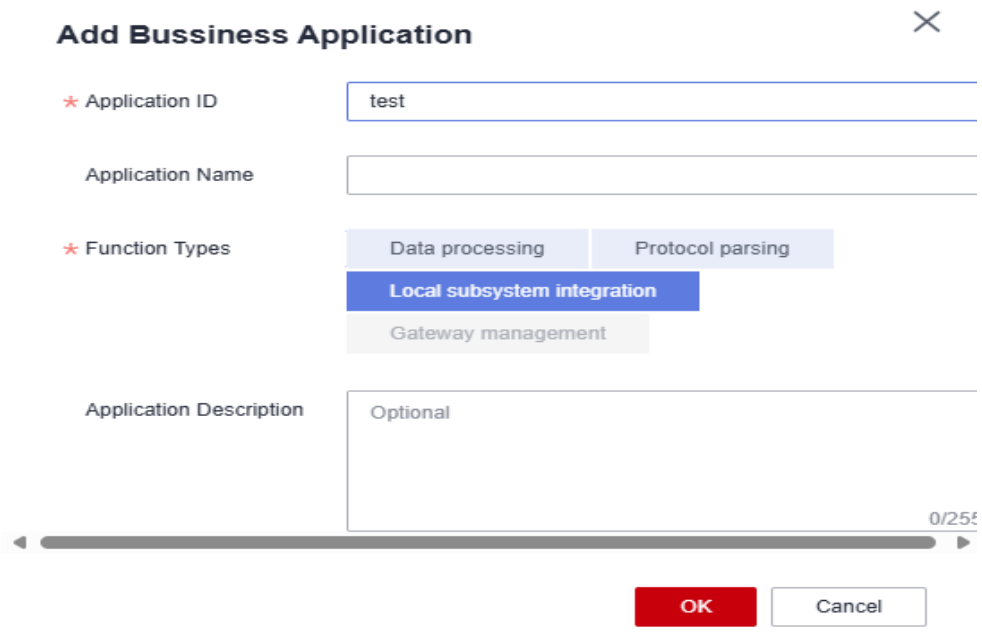
Step 2 Create an application.

1. Choose **IoTEdge** > **Applications** in the left navigation pane. On the page displayed, click **Service Application**, and click **Add Application**.

   For details about how to add a third-party application whose function is local subsystem integration, see the following content.

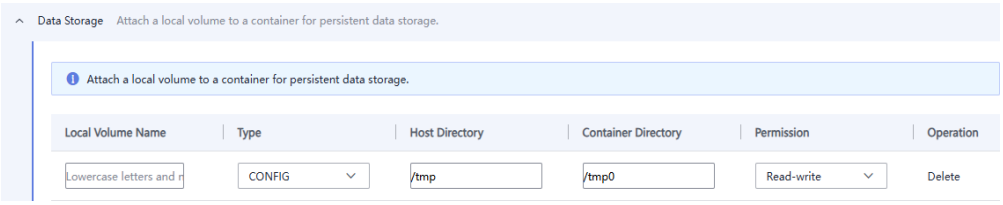
2. When entering application configuration information, set **Function** to **Local subsystem integration**.

Figure 2-204 Application configuration



3. Configure data storage.
   - Set **Type** to **CONFIG**.
   - Set **Permission** to **Read-write**.

Figure 2-205 Data storage



Step 3 Deploy applications.

1. In the navigation pane, choose **IoTEdge** > **Nodes**, click the node bound in **Step 3** to go to the node details page.
2. Choose **Application Module** and click the **Modules** tab to deploy applications.
3. Deploy the edge_apigw application.

   If the edge_apigw application is not deployed on the node, deploy the application first.
4. Deploy the third-party application whose function is local subsystem integration, which is added in **Step 2**.

Step 4 Configure the module.

1. Click the **Module Settings** tab.

   NOTE

   A maximum of 100 configuration items can be added to a configurable module.

2. Click **Add Configuration Item**, set the parameters, and click **OK**.

Table 2-63 Adding configuration items

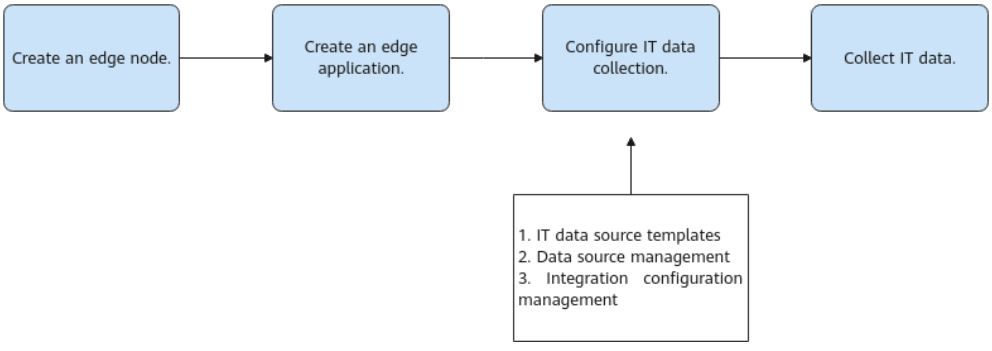
Parameter	Description	Type	Value Range	Example
Configuration Item ID	Unique configuration ID of the southbound IA.	String	Allowed characters: letters, numbers, underscores (_), and hyphens (-) Length: 1 to 64 characters	config001
Configuration Item Name	Name of a configuration item.	String	Allowed characters: letters, numbers, and hyphens (-) Length: 1 to 64 characters	Configuration001
Configuration Value	Content that the edge node obtains from the cloud and sends to the device application.	String	Length: 1 to 2,097,152 characters (within 2 MB). No verification is performed for empty strings.	{"name":"xiaoming","age":18}
Description	Description of the configuration item.	String	Length: 0 to 255 characters	N/A

----End

2.11.4.4 IT Data Collection

IoTEdge collects data of IoT devices and subsystems. IoTEdge collects and delivers different subsystem data based on IT data source configuration.

Figure 2-206 Process for collecting IT data



Prerequisites

IT data collection is available only when the edge node to configure meets the following conditions:

- The **edge_apigw** application and industry data collection application (such as **$industry_dc_bsi**) are deployed on the edge node.

Step 1 Log in to the IoTDA console.

Step 2 In the navigation pane, choose **IoTEdge** > **Nodes**, click the node bound in **Step 3** to go to the node details page.

Step 3 Deploy the **$edge_apigw** application and then the **$industry_dc_bsi** application. Choose **Application Module** > **Modules** and click **Deploy Application**.

----End

Managing IT Data Collection Templates

A data source template describes the metadata of data source configuration items. Before configuring a data source, import a data source template. IoTEdge provides a preset common data source template. You can use the preset template or customize a template.

Viewing a data source template

Step 1 Log in to the IoTDA console. In the navigation pane, choose **IoTEdge** > **Nodes**. On the page displayed, click the created edge node to access its details page.

Step 2 Choose **Application Module** > **Data Configuration** > **Data Source Template Management** to view the preset data source template.

----End

Managing Data Sources

The data source is unique on an edge node and is an instance of the data source template. Systems that provide IT data include the supplier relationship management (SRM), Enterprise resource planning (ERP), manufacturing execution systems (MES), and database.

The following describes how to add two data sources.

Step 1 In the navigation pane, choose **IoTEdge**, click the node bound in **Step 3** to go to the node details page.

Step 2 Choose **Application Module** > **Data Configuration** > **Add Data Source**, add the first data source ERP-API, and click **Save and Deliver**. The platform delivers the new data source to the IT subsystem.

- Data Source ID: ERP-API
- Data Source Name: ERP-API
- Data Source Template: Common Data Source - API
- Service URL: http://120.46.132.171:9000

Step 3 Add the second data source MES-API and click **Save and Deliver**. The platform delivers the new data source to the edge.

- Data Source ID: MES-API
- Data Source Name: MES-API
- Data Source Template: Common Data Source - API
- Service URL: https://dlp-uatbe.mgm-iot.com

NOTE

If you click **Save and Deliver**, the platform delivers the new data source to the edge.

If you click **OK**, the configured data source is only saved on the cloud platform and is not delivered to the edge. You can complete data source delivery operations in the data source list.

----End

Configuring the Integration

An integration configuration template describes the integration configuration for connecting to a specific IT system. Before performing integration configuration, import an integration configuration template.

The following describes how to import a local integration configuration template.

Creating the integrated configuration

Step 1 In the navigation pane, choose **IoTEdge** > **Nodes**, click the node bound in **Step 3** to go to the node details page.

Step 2 Choose **Application Module** > **Data Configuration** > **Integration Configuration**, and click **Add Integration Configuration**. After the configuration is complete, click **Save and Deliver**. The platform delivers data to the MES subsystem synchronously.

- Configuration ID: MES-ERP-WorkPlan
- Upload Configuration Template: Save the code in the **ERP-MES-template.json sample** as a JSON file to the local PC, and upload the file.
- MES Demo (YZ): MES-API
- ERP Demo: ERP-API
- YZtoken Request Header: Retain the default value.
- Product Synchronization Period (min): Retain the default value.
- Synchronization Period (min): Retain the default value.
- Modify: Retain the default item.

ERP-MES-template.json sample

```
{
   "config_values": [
      {
         "crypted": false,
         "name": "YZtoken Header",
         "value": "{\"Authorization\":\"Basic *******=\",\"Content-Type\":\"application/x-www-form-urlencoded
\"}",
         "key": "YZtoken Header"
      },
      {
         "crypted": false,
         "name": "product sync cycle (minutes)",
         "value": "14400",
```



```
            "key": "product sync cycle (minutes)"
        },
        {
            "crypted": false,
            "name": "sync cycle (minutes)",
            "value": "14400",
            "key": "sync cycle (minutes)"
        }
    ],
    "jobs": [
        {
            "cron": "0/20 * * * * ?",
            "output": {
                "scriptContent": "**********",
                "classify": "out",
                "path": "/opendata/v1/erp/production/plan",
                "ds_key": "266",
                "id": 2665,
                "type": "apiCall"
            },
            "input": {
                "scriptContent": "**********",
                "classify": "in",
                "path": "/workingPlan20",
                "ds_key": "267",
                "id": 2664,
                "type": "apiQuery"
            },
            "transform": {
                "scriptContent": "**********",
                "classify": "transform",
                "id": 2666,
                "type": "scriptChange"
            },
            "name": "MES-ProductionPlan-API",
            "id": 2664
        }
    ],
    "name": "MES-ERP",
    "id": 208,
    "data_sources": [
        {
            "name": "MES-Present(YZ)",
            "key": "266"
        },
        {
            "name": "ERP-Present",
            "key": "267"
        }
    ]
}
```


----End

Delivering ERP Data

After the IT data collection configuration is complete, IoTEdge delivers the collected ERP data to the MES subsystem. You can view the data synchronization result on the MES subsystem platform.

Data source delivery

Step 1 In the navigation pane, choose **IoTEdge**, click the node bound in **Step 3** to go to the node details page.

Step 2 Choose **Application Module** > **Data Collection Configuration** > **Data Source**, locate **ERP-API** and **MES-API** in the data source list, and click **Deliver**. Wait for 20

seconds after the delivery status changes to **Delivered**. The platform synchronizes data to the MES subsystem.

----End

Integration configuration delivery

Step 1 In the navigation pane, choose **IoTEdge**, click the node bound in **Step 3** to go to the node details page.

Step 2 Choose **Application Module** > **Data Configuration** > **Integration Configuration**. In the integration configuration list, locate **MES-ERP-WorkPlan** and click **Deliver**. Wait for 20 seconds after the delivery status changes to **Delivered**. The platform synchronizes data to the MES subsystem.

----End

2.11.5 Route Forwarding

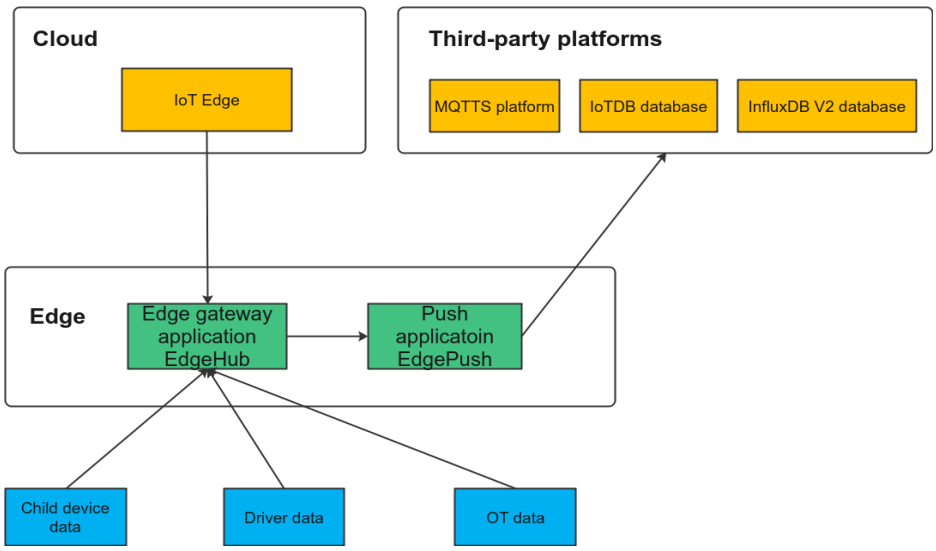
2.11.5.1 Overview

An edge node is a gateway for local devices. It collects edge device data and connects to the third-party cloud platforms and external entities. You can deploy the edge_push application on edge nodes to forward service data to cloud applications or backend services as needed. You can configure the receiver of local device data. Currently, MQTT, IoTDB, and InfluxDB V2 are supported.

NOTICE

Restrictions: Up to 10 channels can be configured for a user. Up to 100 nodes can be allocated to each channel. The maximum channel traffic is 100 TPS.

Figure 2-207 Route forwarding workflow



Currently, child device data, driver data, and OT data can be pushed to third-party MQTTS platforms, IoTDB databases, and InfluxDB V2 databases. For details about the format, see **Procedure**.

Procedure

Step 1 **Create an MQTTS/IoTDB/InfluxDB V2 channel.**

Step 2 **Deploy the edge push application on the node.**

Step 3 **Allocate the channel to the node that requires data push.**

----End

2.11.5.2 Channel Types

2.11.5.2.1 MQTT

The following table lists the parameters that need to be set for an MQTT channel.

Table 2-64 MQTT channel parameters

Parameter	Description
Endpoint Name	Channel name.
Channel	MQTT
Receiving address	Broker address of the third-party MQTT platform. Only TLS channels (for example, **ssl://127.0.0.1:7883**) are supported.
Connection Information	
Authenticate by	Currently, only the user password is supported.
Username	MQTT authentication username.
Password	MQTT authentication password.
Trust Certificate	Whether to verify the domain name when the MQTT broker CA certificate is verified. (If this parameter is deselected, only the server certificate is verified.)
Push Information	
Data Format	Product model (IoTDA 1.0).
Topic	Data push topic (for example, **$oc/devices/gateway**) of the MQTT broker.
QoS	MQTT quality of service (QoS). The default value is **0**.
Description	Channel description.

CAUTION

MQTT client ID cannot be configured. The default value is the node ID.

The data pushed through the MQTT channel is in the IoTDA 1.0 format. The packet format is as follows:

```
{
   "devices": [{
         "device_id": "device1",
         "services": [{
               "service_id": "Motor",
               "properties": {
                     "Current": 10.111,
                     "Voltage": 20
               },
               "event_time": "2006-01-02T15:04:05.000Z"
            }
         ]
      }
   ]
}
```

2.11.5.2.2 IoTDB

The following table lists the parameters that need to be set for an IoTDB channel.

Table 2-65 IoTDB channel parameters

Parameter	Description
Endpoint Name	Channel name.
Channel	Database.
Database Type	IoTDB.
Receiving address	IP address of the third-party IoTDB platform, for example, **127.0.0.1:6667**.
Connection Information	
Username	IoTDB authentication username.
Password	IoTDB authentication password.
Push Information	
Data Format	Product model (IoTDA 1.0).
Storage Group	Storage group to which IoTDB writes data, for example, **edge**. (The corresponding user needs to be authorized to write the storage group in IoTDB.)
Description	Channel description.

In the IoTDB time series database, the fixed prefix of all storage groups is **root.**. For example, if **Storage Group** is set to **edge**, the storage group of the database to which data is written is **root.edge**.

The data written to the database is in the IoTDA 1.0 format. The packet format is as follows:

```
{
   "devices": [{
         "device_id": "device1",
         "services": [{
               "service_id": "Motor",
               "properties": {
                     "Current": 10.111,
                     "Voltage": 20
               },
               "event_time": "2006-01-02T15:04:05.000Z"
            }
         ]
      }
   ]
}
```

The format of the data written to the database is as follows:

```
root.edge.device1.Motor.Current => 10.111
root.edge.device1.Motor.Voltage => 20
```

2.11.5.2.3 InfluxDB V2

The following table lists the parameters that need to be set for an InfluxDB V2 channel.

Table 2-66 InfluxDB V2 channel parameters

Parameter	Description
Endpoint Name	Channel name.
Channel	Database.
Database Type	InfluxDB_V2
Receiving address	Address of the third-party InfluxDB V2 platform, for example, **https://127.0.0.1:8086** or **http://127.0.0.1:8086**.
Connection Information	
Token	Token used by InfluxDB V2 for authentication and identity verification.
Trust certificate	Whether to verify the domain name when the InfluxDB V2 CA certificate is verified. (If this parameter is deselected, only the server certificate is verified.)
Push Information	
Data Format	Product model (IoTDA 1.0).

Parameter	Description
Organization	Organization in InfluxDB V2.
Bucket	Bucket in InfluxDB V2.
Description	Channel description.

Measurement under bucket in InfluxDB V2 uses the data format of the IoTDA 1.0 product model.

For example, the product name is **ElectricalMachinery** and the packet is as follows:

```
{
   "devices": [{
          "device_id": "device1",
          "services": [{
                 "service_id": "Motor",
                 "properties": {
                        "Current": 10.111,
                        "Voltage": 20
                 },
                 "event_time": "2006-01-02T15:04:05.000Z"
            }
          ]
      }
   ]
}
```

When data is written into the database, the product name is used as the measurement, and the device ID and service ID are used as tags. The format of data written to the database is as follows:

```
measurement: ElectricalMachinery
tags: {"device": "device1", "service": "Motor"}  fields: {"Current": 10.111, "Voltage": 20}
```

2.11.5.3 Creating a Channel

Route forwarding allows you to manage data receiving channels, including adding, deleting, modifying, and allocating channels.

Procedure

Step 1 Log in to the IoTDA console.

Step 2 In the navigation pane, choose **IoTEdge** > **Route Forwarding**. Click **Add Data Receiving Endpoint** in the upper right corner and specify the parameters based on the following table.

Currently, MQTT, IoTDB, and InfluxDB V2 channels are supported.

- For details about how to add an MQTT channel, see **MQTT**.
- For details about how to add an IoTDB channel, see **IoTDB**.
- For details about how to add an InfluxDB V2 channel, see **InfluxDB V2**.

Step 3 Click **OK**.

----End

2.11.5.4 Deploying the Edge Push Application on a Node

Procedure

> **CAUTION**
>
> Route forwarding requires EdgeHub 1-1-21 or later.

Step 1 Log in to the IoTDA console.

Step 2 In the navigation pane, choose **IoTEdge** > **Nodes**, and click the target edge node name.

Step 3 Choose **Application Module** and click the **Modules** tab to deploy applications.

Step 4 Select the data processing function, select the **$edge_push** application, select a version, and click **OK**.

----End

2.11.5.5 Allocating a Channel to a Node

Procedure

Step 1 Log in to the IoTDA console.

Step 2 In the navigation pane, choose **IoTEdge** > **Route Forwarding**, click a created MQTTS channel to go to the channel details page, and click **Allocate Node**.

Step 3 On the node allocation page, select the node where the edge push application is deployed in **Deploying the Edge Push Application on a Node** and click **OK**.

Step 4 A message is displayed in the upper right corner, indicating that the allocation is successful.

----End

3 Best Practices

3.1 Connecting a Device Simulator to IoTDA

This topic uses a device simulator as an example to describe how to connect devices to IoTDA using the native MQTT protocol. The device simulator is an MQTT client, which enables you to easily verify whether devices can interact with IoTDA to publish or subscribe to messages.

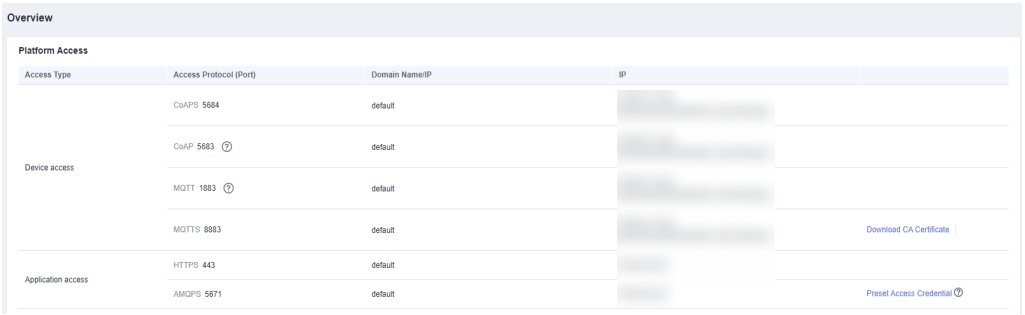
Obtaining Device Access Information

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

Step 1 Log in to the IoTDA console.

Step 2 Choose **Overview** in the navigation pane, view the device access information, and record the IP address and port number.

Step 3 Click **Download CA Certificate** to save the server certificate file.



----End

Creating a Product

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

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

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

Parameter	Description
Resource Space	IoTDA automatically allocates the created product to the default resource space. • If you want to allocate the product to another resource space, select the required resource space from the drop-down list. • If the corresponding resource space does not exist, choose **Resource Spaces** in the left navigation pane and create a resource space.
Product Name	Name the product. The product name is unique in the resource space.
Protocol	Select **MQTT**.
Data Type	Select **JSON**.
Manufacturer	Enter the manufacturer name of the device.
Device Type	You can set it to **StreetLight**, **GasMeter**, or **WaterMeter**.

----End

Registering a Device

Step 1 On the console, choose **Devices** > **All Devices** in the navigation pane, and click **Register Device** in the upper right corner.

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

Parameter	Description
Resource Space	Select the same resource space of the product created in **Creating a Product**.
Product	Select the product created in **Creating a Product**.
Node ID	Customize a unique physical identifier for the device. The device ID (corresponding to the node ID) and secret generated after the registration are used for device access.
Device Name	Customize the device name.
Authentication Type	Select **Secret**.
Secret	Specify a secret used for device access. If no secret is specified, IoTDA automatically generates one.

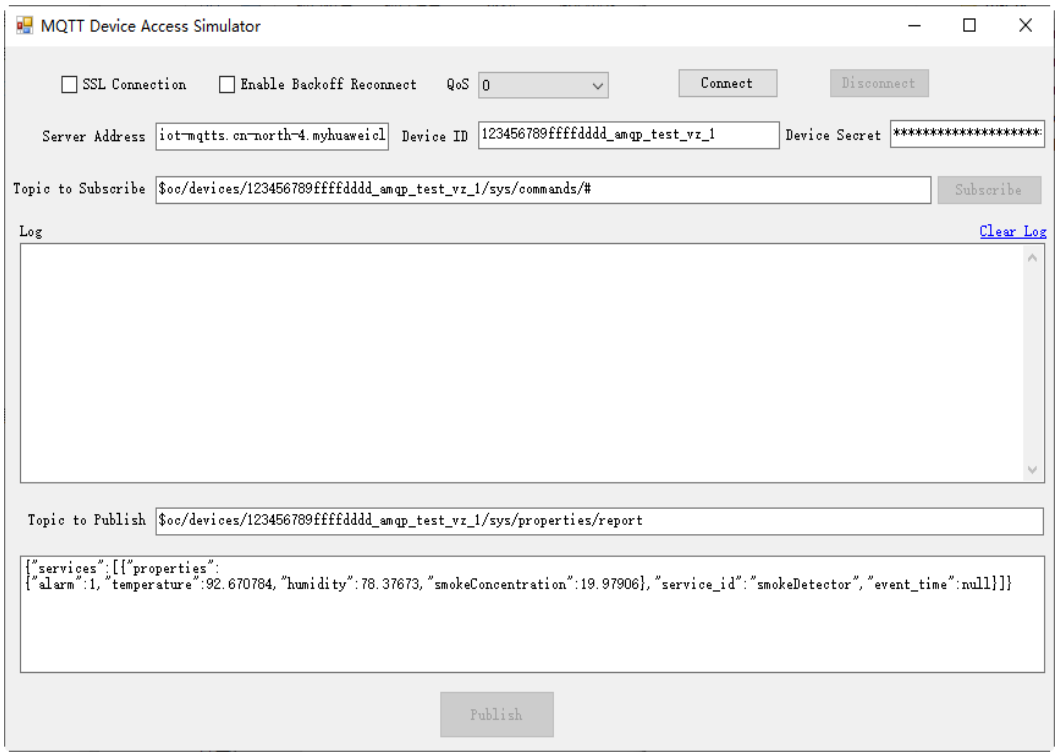
Save the device ID and secret. They are used for authentication when the device attempts to access IoTDA.

----End

Connecting a Device Simulator to IoTDA

Step 1 Download the **simulator** (for 64-bit operating system by default) and start it.

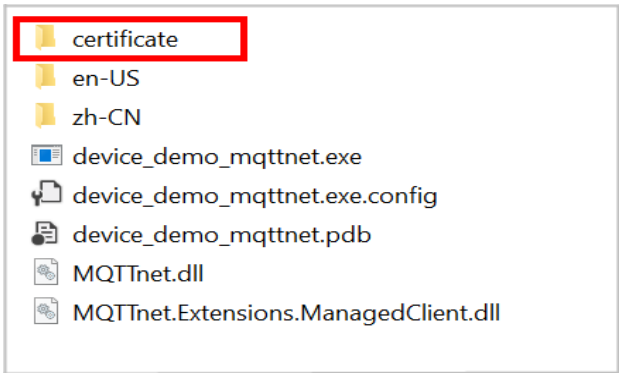
Step 2 Perform operations on the UI.

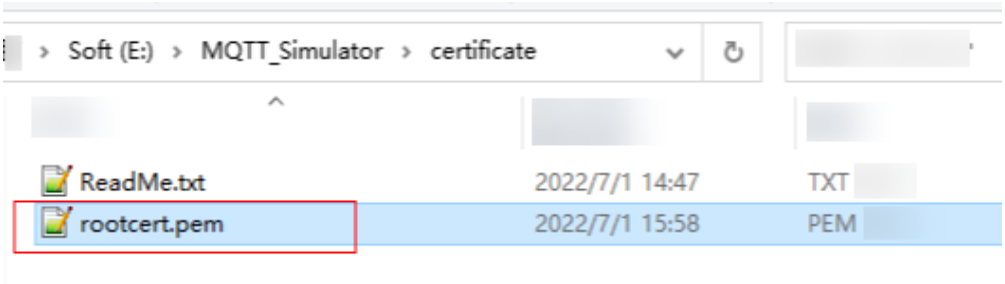


1. On the simulator UI, enter the server address, device ID, and device secret. Set the parameters based on the actual device information.

- Server address: domain name. For details about how to obtain it, see "Developer Guide" > "Obtaining Resources" > "Platform Connection Information" in the *Usage Guide*.
- Device ID and secret: Obtain them from **Registering a Device**.

2. Use the corresponding certificate files together with different server addresses during SSL-encrypted access. Obtain a certificate by referring to **Obtaining Device Access Information**. Replace the certificate in the **certificate** folder and change the certificate name to **rootcert.pem**, as shown in the following figures.

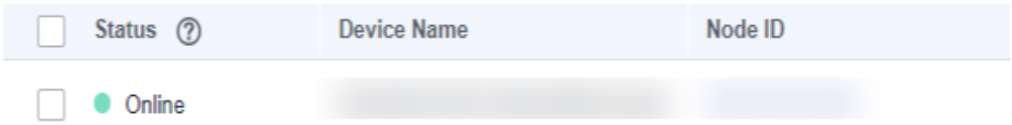




3. Select SSL encryption or no encryption when establishing a connection on the device side and set the QoS mode to 0 or 1. Currently, QoS 2 is not supported. For details, see "Service Overview" > "Limitations" in the *Usage Guide*.

Step 3 Establish a connection.

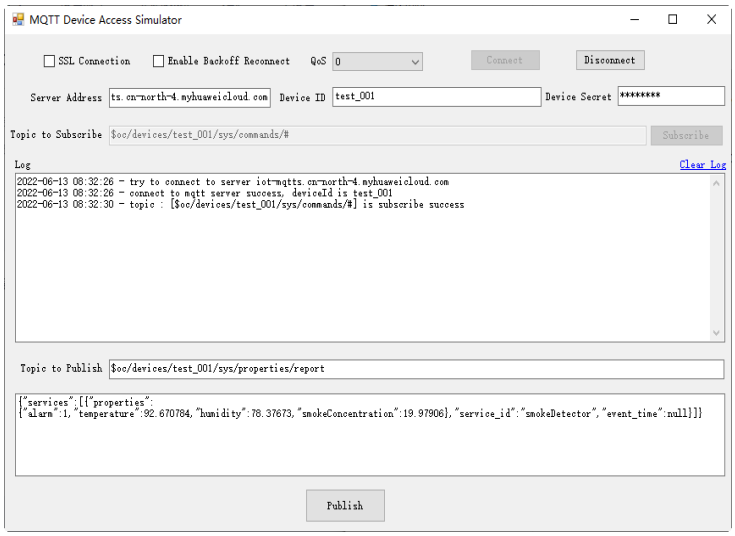
To connect a device or gateway to IoTDA, upload the device information to bind the device or gateway to IoTDA. Click **Connect**. If the domain name, device ID, and secret are correct, a device connection success is displayed in the log. Check the device status on IoTDA, as shown in the following figure.



Step 4 Subscribe to a topic.

Only devices that subscribe to a specific topic can receive messages about the topic published by the broker. For details on the preset topics, see "API Reference" > "API Reference on the Device Side" > "Topics" in the *Usage Guide*.

After the connection is established and a topic is subscribed, the following information is displayed in the log area on the home page of the demo:

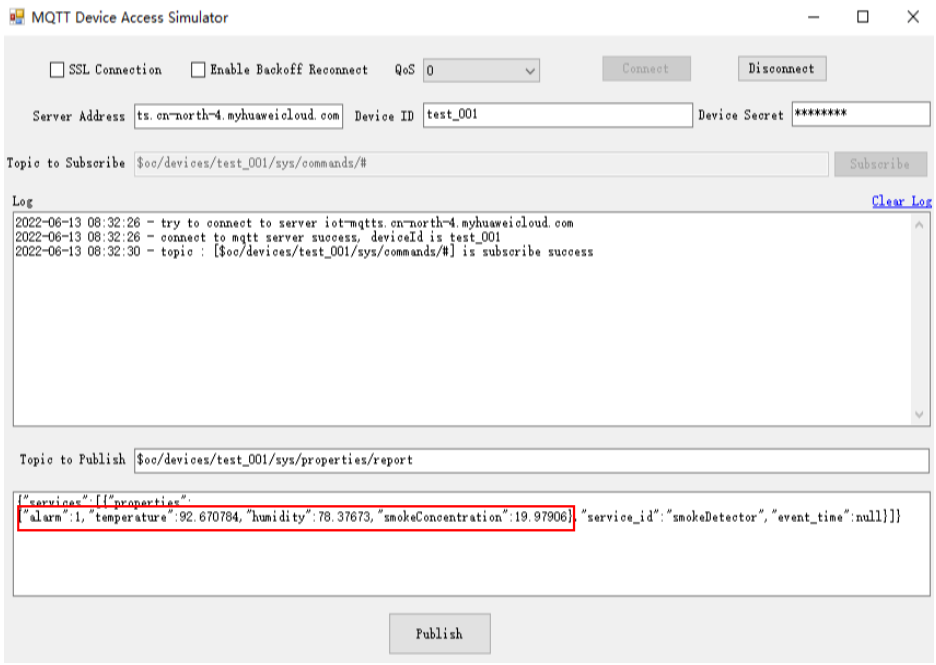


Step 5 Publish a topic.

Publishing a topic means that a device proactively reports its properties or messages to IoTDA. For details, see "API Reference" > "API Reference on the Device Side" > "Device Property APIs"> "Reporting Device Properties" in the *Usage Guide*.

The simulator implements the property reporting topic and property reporting.

After a topic is published, the following information is displayed on the demo page:



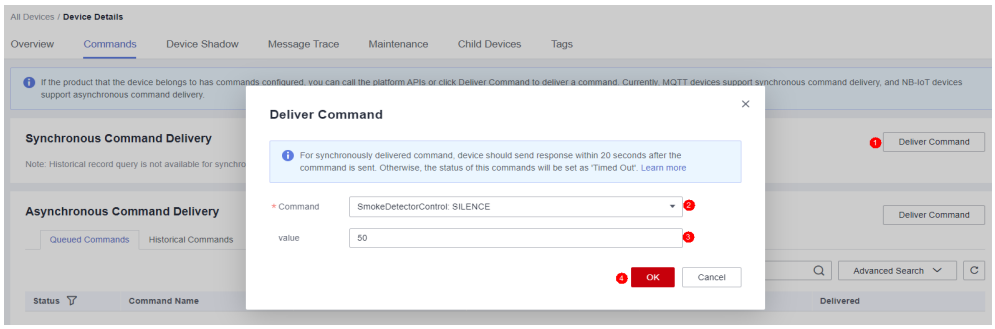
If the reporting is successful, the reported device properties are displayed on the device details page.

Step 6 Receive a command.

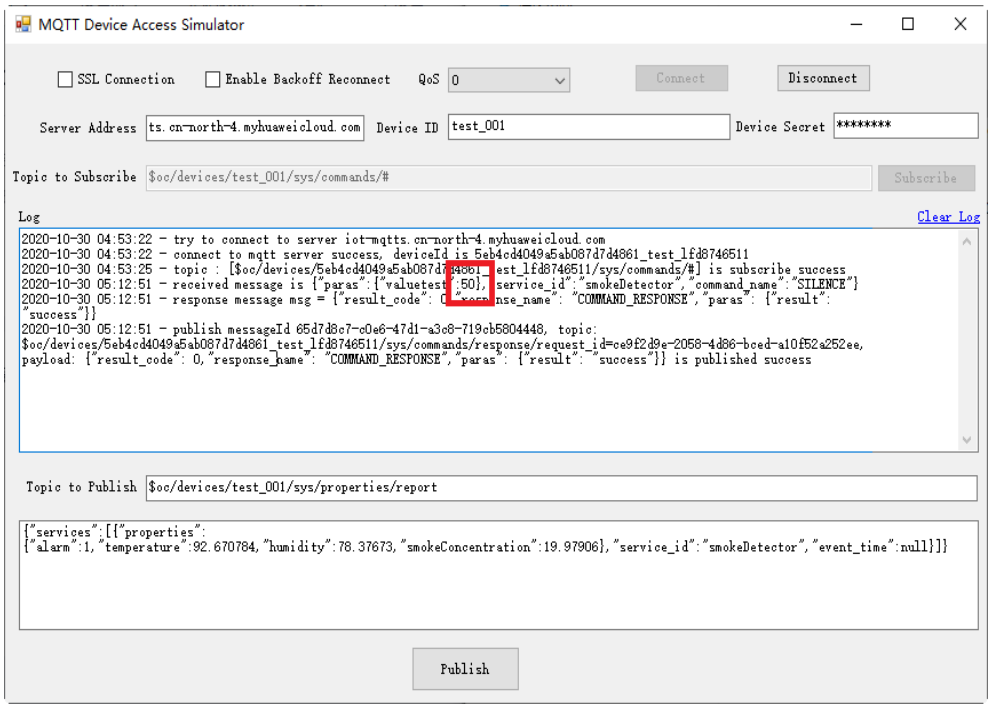
The simulator can receive commands delivered by the platform. After an MQTT connection is established and a topic is subscribed, you can deliver a command on the device details page of the IoTDA console. After the command is delivered, the MQTT callback receives the command delivered by the platform.

In the navigation pane, choose **Products** and click the target product. On the **Model Definition** page, click **Add Command**, set the command name and parameters, and click **OK** to save the command.

In the navigation pane, choose **Devices** > **All Devices** > **View**. On the device details page, choose **Commands** > **Synchronous Command Delivery** > **Deliver Command**. For example, deliver the command whose parameter name is **smokeDetector: SILENCE** and parameter value is **50**.



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



----End

3.2 Automatic Device Shutdown Upon High Temperature

Scenarios

IoTDA supports device data reporting and command delivery. To associate the two, an application needs to provide corresponding logic.

However, with the rule function provided by IoTDA, the platform can automatically deliver specified commands when specific data is reported, reducing the application development workload.

In this example, when the temperature reported by the temperature sensor of a device is higher than 80°C, IoTDA automatically delivers a command to shut down the device.

Configuring IoTDA

Using IoTDA, you can create a product, upload a model file, register a device, and set a device linkage rule to enable IoTDA to send a command when receiving specific data from the device.

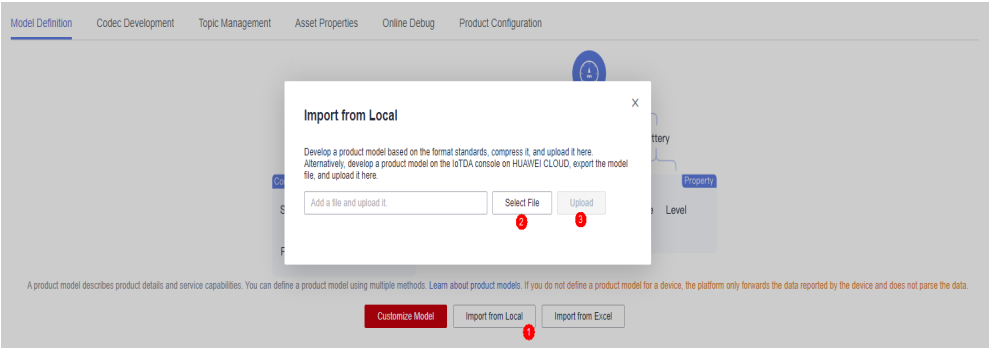
1. Log in to the IoTDA console.
2. Choose **Products** in the navigation pane and click **Create Product** in the upper right corner to create an MQTT product. Set the parameters and click **OK**.

   Note: The product model and device used in this topic are only examples. You can use your own product model and device.

Parameter	Description
Resource Space	IoTDA automatically allocates the created product to the default resource space. • If you want to allocate the product to another resource space, select the required resource space from the drop-down list. • If the corresponding resource space does not exist, choose **Resource Spaces** in the left navigation pane and create a resource space.
Product Name	Name the product. The product name is unique in the resource space.
Protocol	Select **MQTT**.
Data Type	Select **JSON**.
Manufacturer	Enter the manufacturer name of the device.
Device Type	Set this parameter based on service requirements.

3. Click **Profile.zip** to download a sample product model file.
4. Click the created product to go to the product details page. On the **Model Definition** tab page, click **Import from Local**. In the displayed dialog box, click **Select File**, select the product model file obtained in the previous step, and click **Upload**.

Figure 3-1 Uploading a model file



5. In the navigation pane, choose **Devices** > **All Devices**. Click **Register Device** and configure the device registration parameters.

Parameter	Description
Product	Select the product created in 2.
Node ID	Set this parameter to the IMEI, MAC address, or serial number of the device. If the device is not a physical one, set this parameter to a custom character string that contains letters and digits.

Parameter	Description
Device Name	Customize the value.
Authentication Type	Select **Secret**.
Secret	Specify a secret used for device access. If no secret is specified, IoTDA automatically generates one.

Click **OK**. Click the button to save the device ID and secret returned after the registration is successful.

6. In the navigation pane, choose **Rules** > **Device Linkage**, and click **Create Rule** in the upper right corner.
7. Set the parameters based on the table below. The following parameter values are only examples. You can create your own rules. After setting the parameters, click **Create Rule**.

Parameter	Description
Rule Name	Specify the name of the rule to be created, for example, **Overheated**.
Activation	Select **Activate upon creation**.
Effective Period	Select **Always effective**.
Description	Provide a description of the rule, for example, "The device is automatically shut down when the device temperature is higher than 80°C."
Set Triggers	1. Select **Triggered upon specified device**. 2. Select the device added in **5**. 3. Select **tempSensor** for **Select service**, **temperature** for **Select property**, **>** as the operation, and enter **80**. Click **Trigger Mode**. In the displayed dialog box, set **Data Validity Period (s)** to **300** and click **OK**.
Set Actions	1. Select **Deliver Commands**, and select the device created in **5**. 2. Select **deviceSwitch** for **Select service**, and **ON_OFF** for **Select command**. Click **Configure Parameter**. In the dialog box displayed, set **power** to **OFF**, and click **OK**.

Verifying the Configurations

- You can use a registered physical device to access the platform and enable the device to report the temperature greater than 80.

- You can also use a simulator to simulate a device to publish the topic **$oc/devices/***{device_id}*/**sys/properties/report** (replace {*{device_id}*} with the actual device ID) and report data whose temperature is greater than 80.

Expected result:

- If you use a physical device to report data, the device receives an ON_OFF command in which **power** is **OFF**.
- If you use a simulator to report data, subscribe to Topic: **$oc/devices/***{device_id}*/**sys/commands/#** before reporting data. Replace *{device_id}* with the actual device ID. After data is reported, the **ON_OFF** command with the **power** parameter set to **OFF** is displayed on the **Subscribe** tab page.

3.3 Triggering and Forwarding an Alarm

Scenarios

IoTDA supports device alarms. You can configure rules to push alarms to applications.

Device alarms can be generated in either of the following ways:

1. Device linkage rule: When a device reports data and meets the linkage rule triggering condition, a device alarm is generated.
2. Device alarm API: A device directly reports alarms through APIs. For details about the topic and the alarm data format, see "API Reference on the Device Side" > "Device Alarm Management APIs" in the *API Reference*.

This section describes the first mode. Based on the configured device linkage rule, when the temperature reported by the device sensor is higher than 80°C, a device alarm is generated and then pushed to the application.

Configuring IoTDA

You can create a product, upload a model file, register a device, set a device linkage rule, and configure a data forwarding rule. In this way, when a device reports specific data, the linkage rule is triggered and an alarm is generated.

Step 1 Log in to the IoTDA console.

Step 2 Choose **Products** in the navigation pane and click **Create Product** in the upper right corner to create an MQTT product. Set the parameters and click **OK**.

CAUTION

The product model and device used in this topic are only examples. You can use your own product model and device.

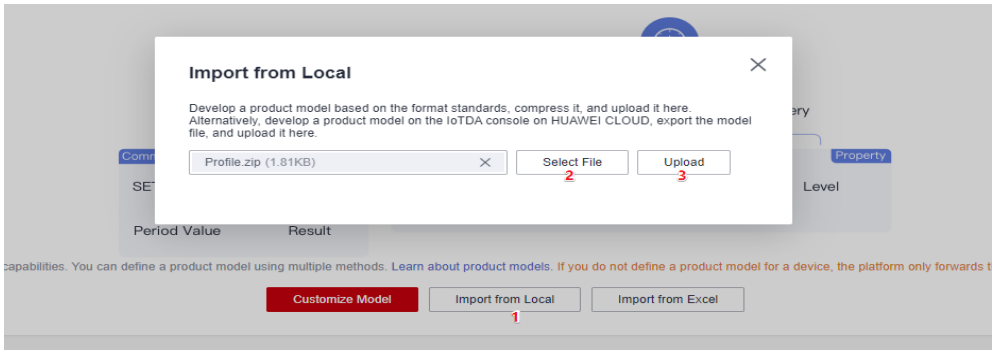
Table 3-1 Creating a product

Parameter	Description
Resource Space	IoTDA automatically allocates the created product to the default resource space. • If you want to allocate the product to another resource space, select the required resource space from the drop-down list. • If the corresponding resource space does not exist, choose **Resource Spaces** in the left navigation pane and create a resource space.
Product Name	Name the product. The product name is unique in the resource space.
Protocol	Select **MQTT**.
Data Type	Select **JSON**.
Manufacturer	Enter the manufacturer name of the device.
Device Type	Set this parameter based on service requirements.

Step 3 Click **Profile.zip** to download a sample product model file.

Step 4 Click the created product to go to the product details page. On the **Model Definition** tab page, click **Import from Local**. In the displayed dialog box, click **Select File**, select the product model file obtained in the previous step, and click **Upload**.

Figure 3-2 Uploading a model file



Step 5 In the navigation pane, choose **Devices** > **All Devices**. Click **Register Device** and configure the device registration parameters.

Table 3-2 Registering a device

Parameter	Description
Product	Select the product created in **2**.

Parameter	Description
Node ID	Set this parameter to the IMEI, MAC address, or serial number of the device. If the device is not a physical one, set this parameter to a custom character string that contains letters and digits.
Device Name	Customize the value.
Authentication Type	Select **Secret**.
Secret	Specify a secret used for device access. If no secret is specified, IoTDA automatically generates one.

Click **OK**. Click the button to save the device ID and secret returned after the registration is successful.

Step 6 In the navigation pane, choose **Rules** > **Device Linkage**, and click **Create Rule** in the upper right corner.

Step 7 Set the parameters based on the table below. The following parameter values are only examples. You can create your own rules. After setting the parameters, click **Create Rule**.

Table 3-3 Creating a linkage rule

Parameter	Description
Rule Name	Specify the name of the rule to be created, for example, **Overheated**.
Activation	Select **Activate upon creation**.
Effective Period	Select **Always effective**.
Description	Provide a description of the rule, for example, "The device is automatically shut down when the device temperature is higher than 80°C."
Set Triggers	1. Select **Triggered upon specified device**. 2. Select the device added in **5**. 3. Select **tempSensor** for **Select service**, **temperature** for **Select property**, **>** as the operation, and enter **80**. Click **Trigger Mode**. In the displayed dialog box, set **Data Validity Period (s)** to **300** and click **OK**.
Set Actions	Select **Report alarms** for the action type, select the alarm severity, enter the alarm name, and edit the alarm content.

Step 8 In the navigation pane, choose **Rules** > **Data Forwarding**, and click **Create Rule** in the upper right corner.

Step 9 Set the parameters based on the table below. The following parameter values are only examples. You can create your own rules. After setting the parameters, click **Create Rule**.

Figure 3-3 Creating a rule

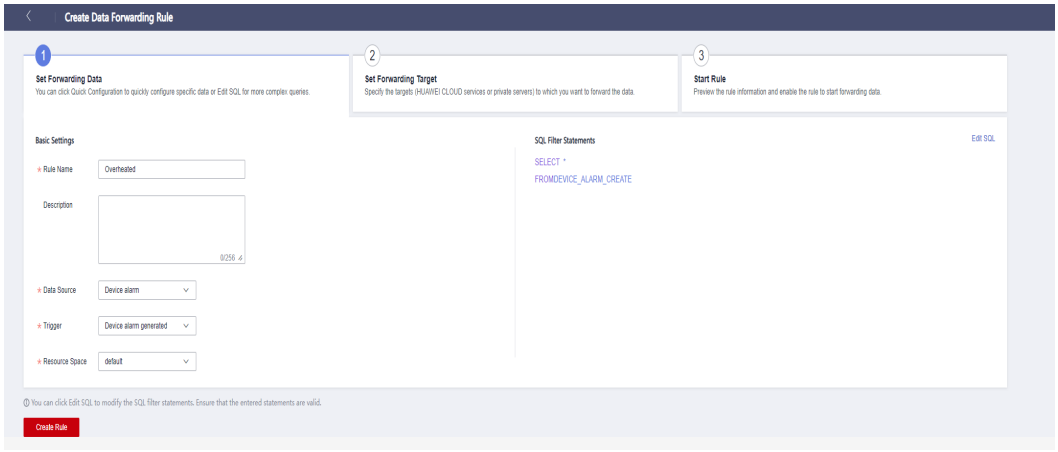
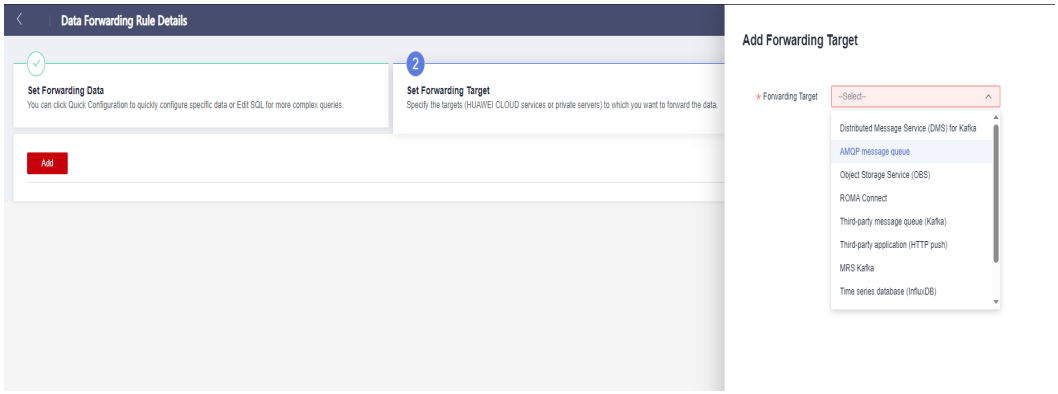


Table 3-4 Creating a rule

Parameter	Description
Rule Name	Specify the name of the rule to be created, for example, **Overheated**.
Description	Describe the rule.
Data Source	Select **Device alarm**.
Trigger	Select **Device alarm generated**. (You can also select **Device alarm cleared** to push an alarm clearance notification.)
Resource Space	Select the resource space to which the product belongs.

Step 10 Configure the forwarding target. Select the target type, configure related information, and enable the rule.

Figure 3-4 Setting the forwarding target



----End

Verifying the Configurations

- You can use a registered physical device to access the platform and enable the device to report the temperature greater than 80.
- You can also use a simulator to simulate a device to publish the topic **$oc/devices/***{device_id}*/**sys/properties/report** (replace {*{device_id}*} with the actual device ID) and report data whose temperature is greater than 80.

Expected result:

- The configured forwarding target can receive the device alarm notification pushed by the platform.

3.4 Sharing Device Location Information

Scenarios

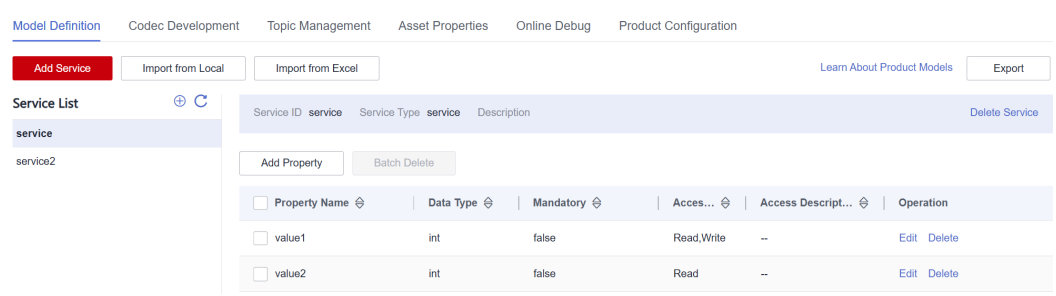
You can set data forwarding rules to push the device geographical information to applications. When devices report properties or you configure device asset properties and tags, the rules are triggered and the platform pushes data to applications.

Data Sharing Through Device Properties

Step 1 Access the IoTDA console, create an MQTT product, add a device, and connect the device to the platform. For details, see **Connecting a Device Simulator to IoTDA**.

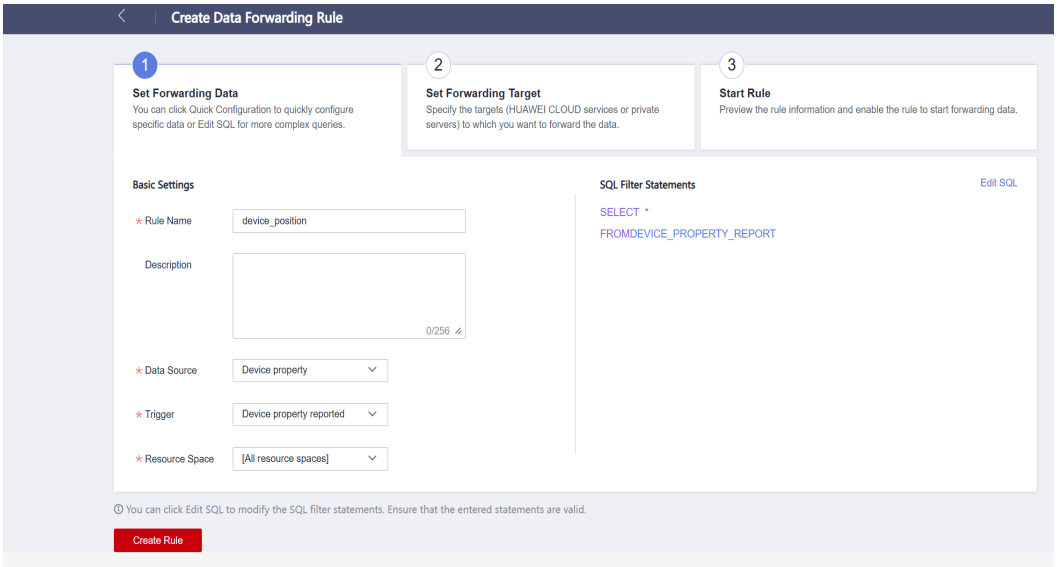
Step 2 Customize the product model and add the **longitude** and **latitude** properties.

Figure 3-5 Defining a product model



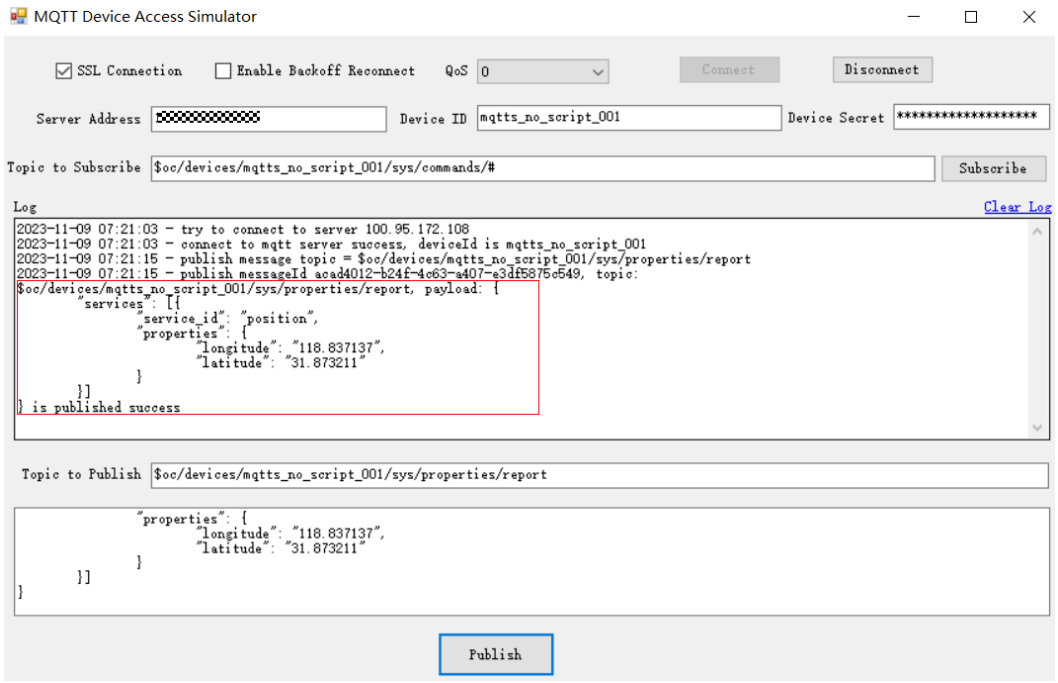
Step 3 Create a rule for device property data forwarding. For details, see "Rules" > "Data Forwarding" in the *User Guide*.

Figure 3-6 Creating a rule



Step 4 The device reports the longitude and latitude information to the platform through property reporting.

Figure 3-7 Reporting properties



Step 5 The platform receives the data and the rule is triggered to forward the data to the application.

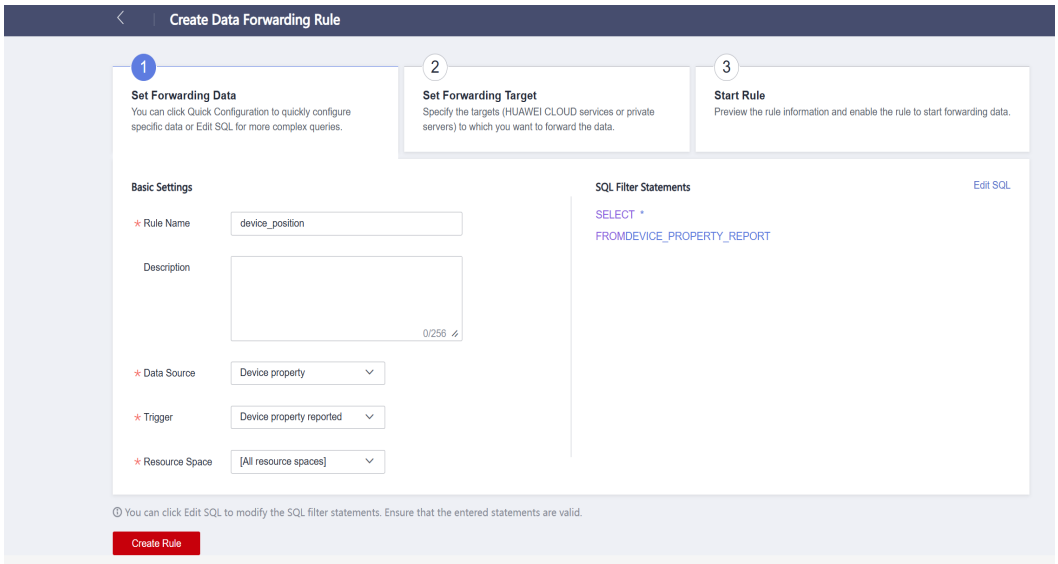
----End

Data Sharing Through Tags

Step 1 Access the IoTDA console, create an MQTT product, add a device, and connect the device to the platform. For details, see **Connecting a Device Simulator to IoTDA**.

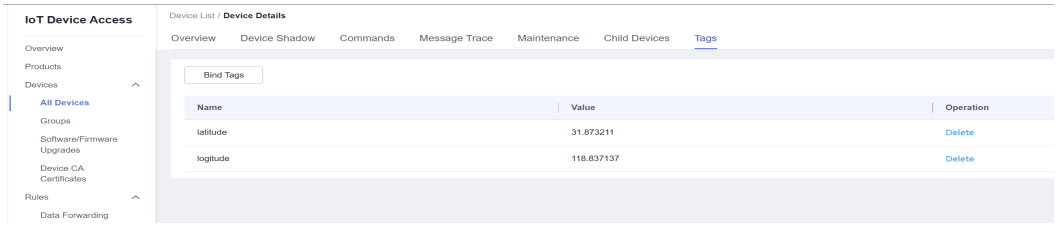
Step 2 Create a rule for device property data forwarding. For details, see "Rules" > "Data Forwarding" in the *User Guide*.

Figure 3-8 Creating a rule



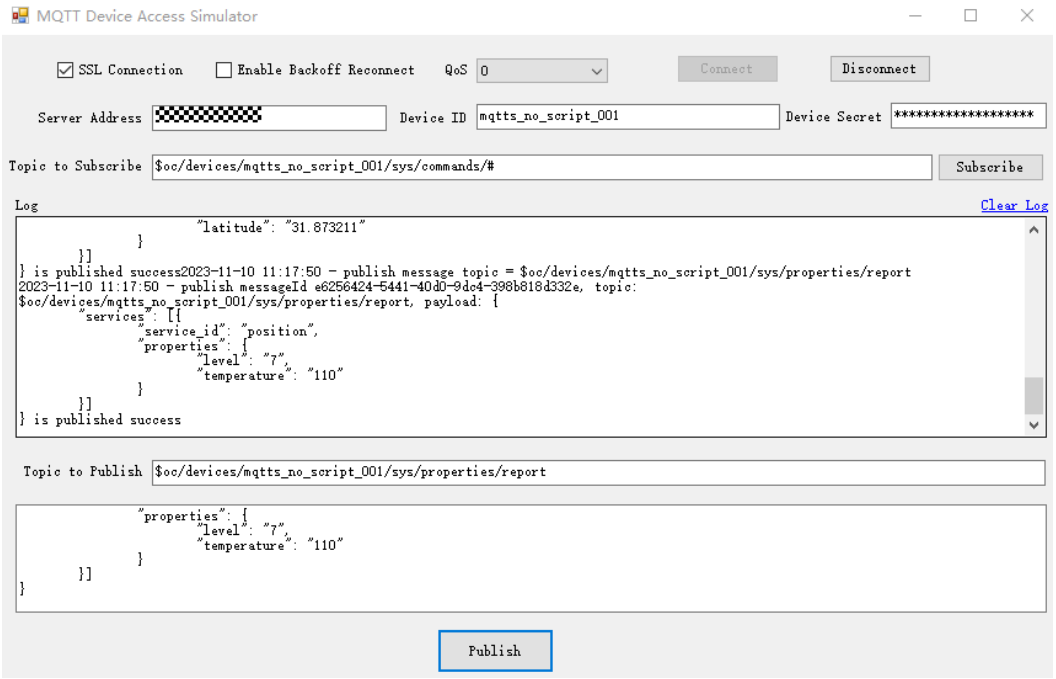
Step 3 On the device details page, click the **Tags** tab to add the longitude and latitude tags.

Figure 3-9 Adding a tag



Step 4 When the device reports any property data and triggers the data forwarding rule, the data is pushed to the application together with the tag information.

Figure 3-10 Reporting properties



Data received by the application:

```
{
   "resource": "device.property",
   "event": "report",
   "event_time": "20231110T031806.148Z",
   "notify_data": {
      "header": {
         "app_id": "39a2781036184a4496f58528577f4954",
         "device_id": "mqtts_no_script_001",
         "node_id": "mqtts_no_script_001",
         "product_id": "653f70759ca2f949659efea0",
         "gateway_id": "mqtts_no_script_001",
         "tags": [{
            "tag_key": "latitude",
            "tag_value": "31.873211"
         }, {
            "tag_key": "longitude",
            "tag_value": "118.837137"
         }]
```



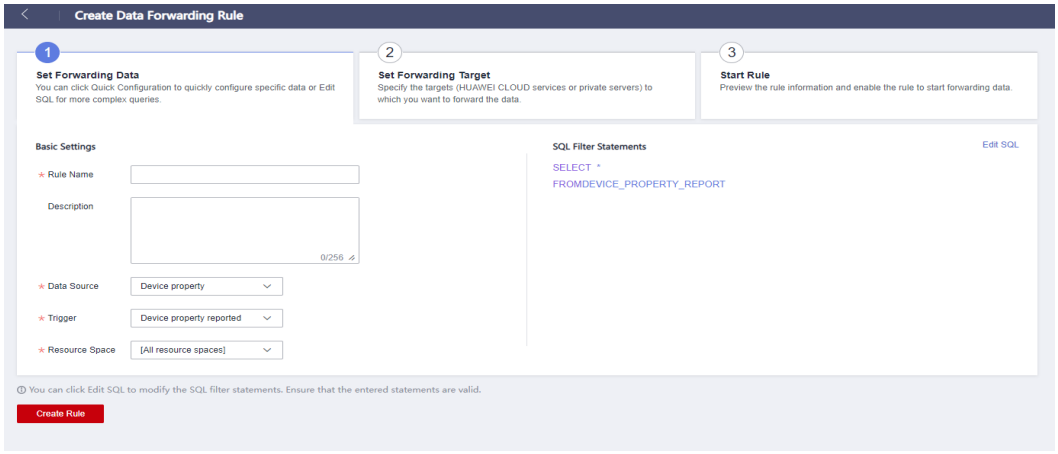
```
        },
        "body": {
            "services": [{
                "service_id": "position",
                "properties": {
                    "level": "7",
                    "temperature": "110"
                },
                "event_time": "20231110T031806Z"
            }]
        }
    }
}
```

----End

Data Sharing Through Asset Properties

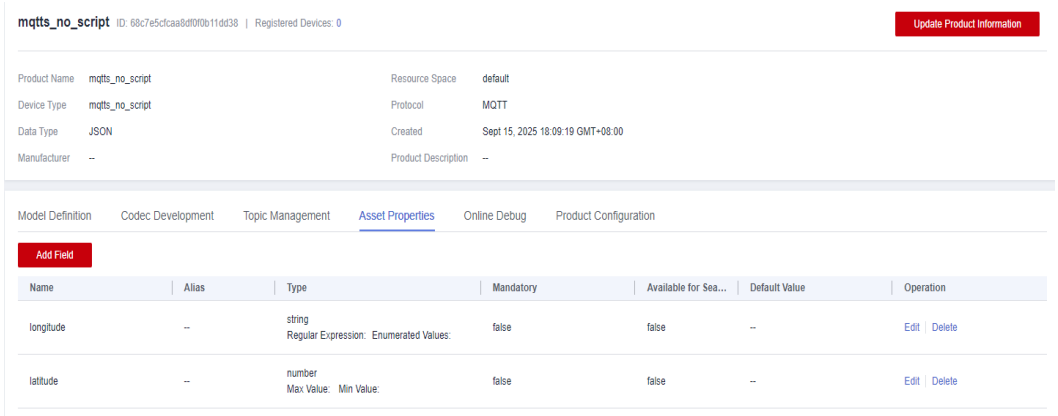
Step 1 Create a data forwarding rule by referring to "Rules" > "Data Forwarding" in the *User Guide*. Select **Device** for **Data Source**, **Device updated** or **Device added** for **Trigger**, and select the resource space to which the product belongs.

Figure 3-11 Creating a rule



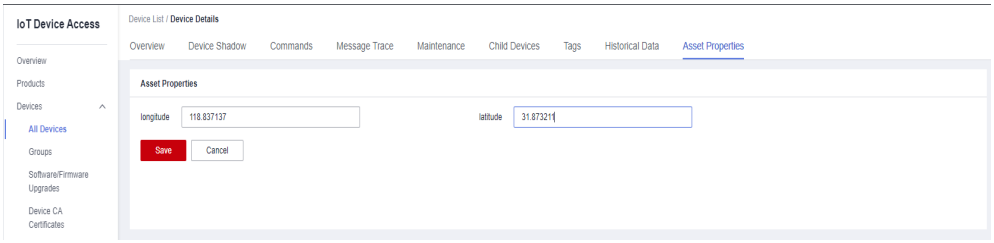
Step 2 On the product details page, click the **Asset Properties** tab and add the **longitude** and **latitude** fields.

Figure 3-12 Configuring product asset properties



Step 3 On the device details page, click the **Asset Properties** tab, configure asset properties, and click **Save**. The rule created in **Step 1** is triggered, and the asset properties are pushed to the forwarding target.

Figure 3-13 Configuring device asset properties



----End

3.5 Using a Custom Topic for Communication

Scenarios

You can customize topics for MQTT devices in device message reporting and platform message delivery. Applications can implement different service logic based on topics. Custom topics can also be used in the scenario where a device cannot report properties or receive delivered commands defined in the product model.

In this example, an application receives the temperature data reported by a device through the topic and determines whether to turn on or off the indoor air conditioner.

Prerequisites

- You have created an MQTT product, developed a product model, and added a device on the IoTDA console. For details, see **Connecting a Device Simulator to IoTDA**.

Message Reporting

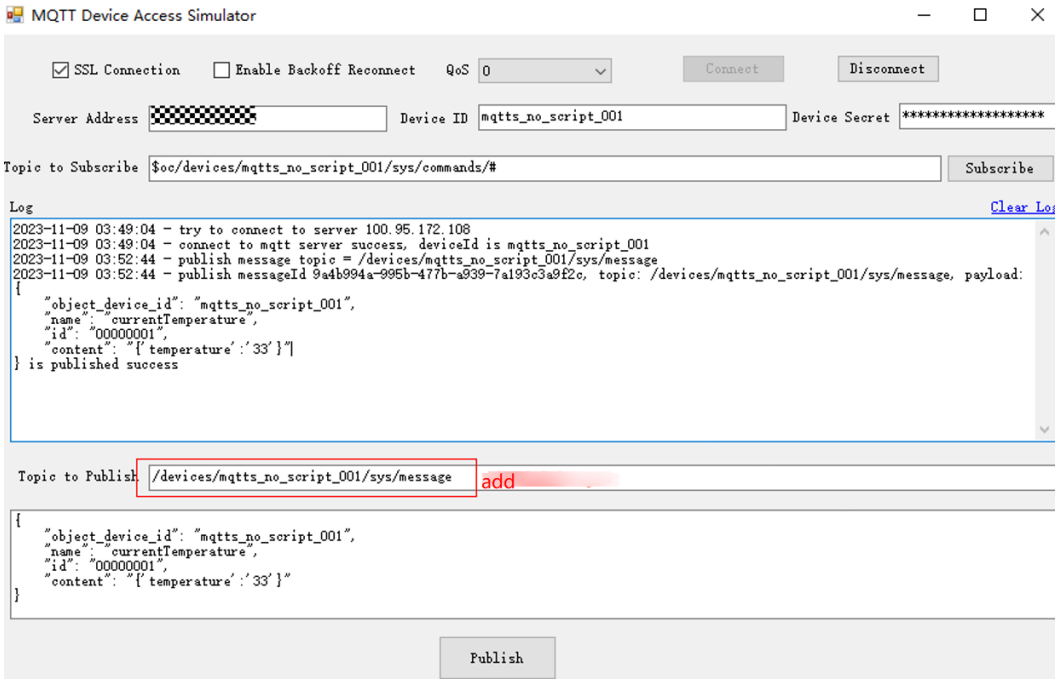
Step 1 Access the IoTDA console.

Step 2 In the navigation pane, choose **Devices** > **All Devices**, find the corresponding device, and click **View** to access its details page.

Step 3 Click the **Message Trace** tab, click **Start Trace**, and set the trace duration as required.

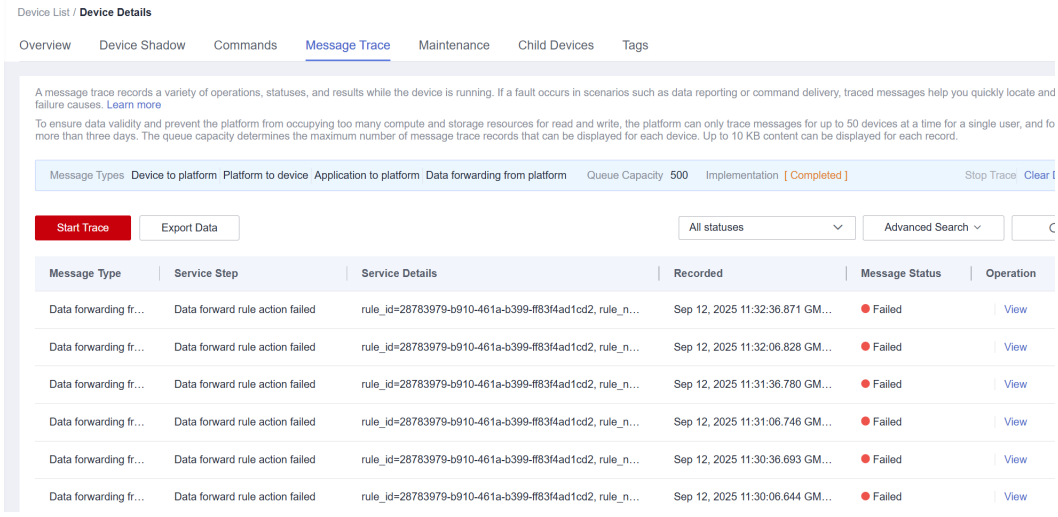
Step 4 Simulate a device to report a custom topic message. For details, see **Connecting a Device Simulator to IoTDA**.

Figure 3-14 Reporting messages through custom topics



Step 5 On the **Message Trace** page, view the custom topic message reported by the device.

Figure 3-15 Message tracing



Step 6 An application obtains the custom topic message reported by the device through data forwarding. For details about data forwarding modes, see "Rules" > "Data Forwarding" in the *User Guide*.

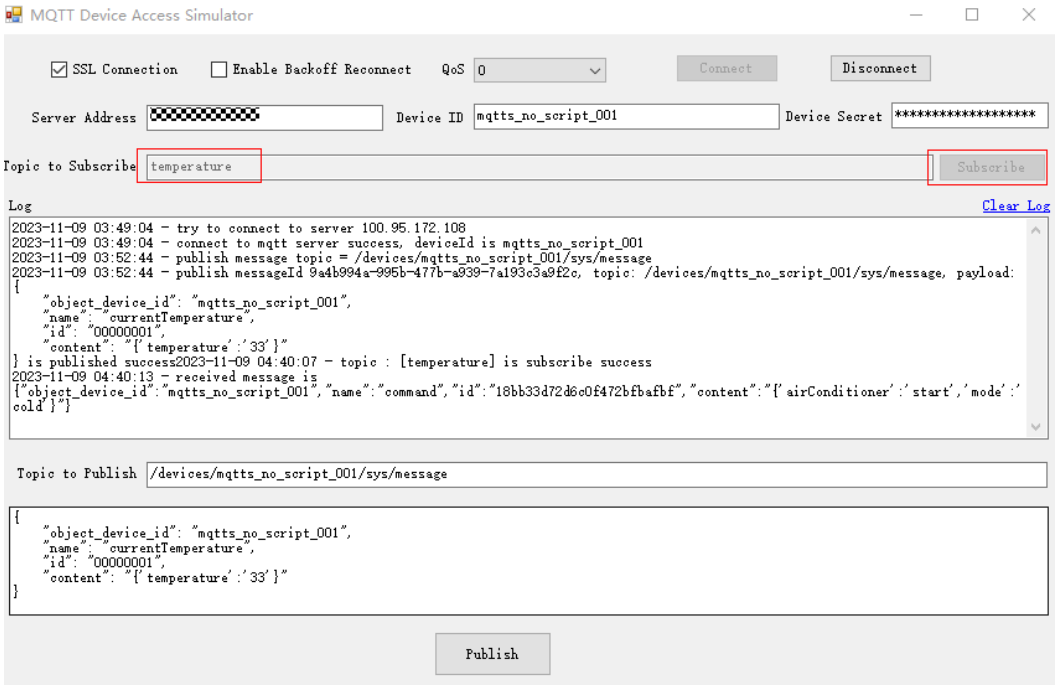
----End

Message Delivery

In this example, the Postman tool is used to deliver an instruction for turning on the indoor air conditioner.

Step 1 Use the MQTT simulator to simulate a device to subscribe to a custom topic.

Figure 3-16 Subscribing to a topic

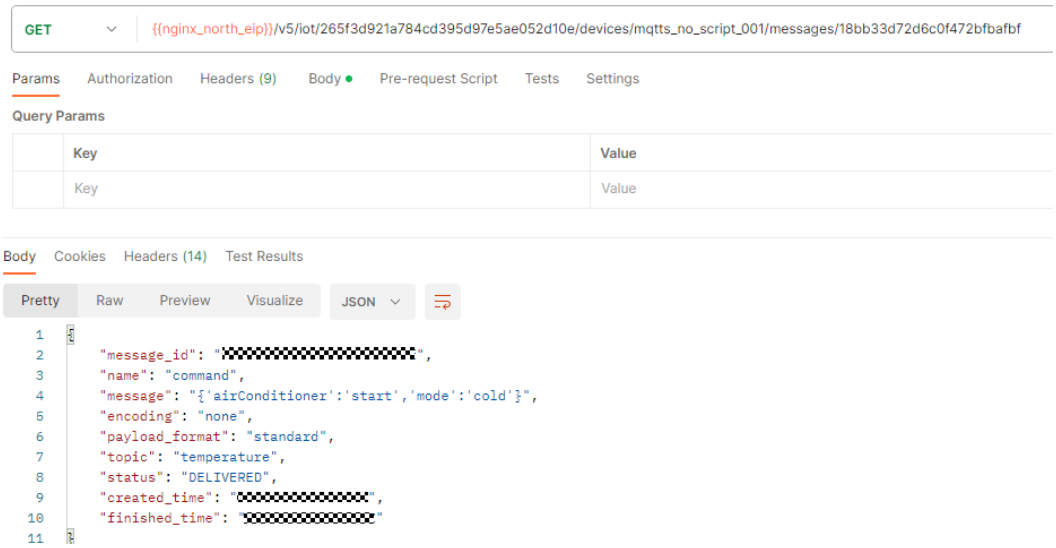


Step 2 Use the Postman tool to simulate an application to call the API for message delivery to deliver an instruction for turning on the indoor air conditioner.

Figure 3-17 Calling the message delivery API

Step 3 Call the API for querying device messages to check whether the instruction is delivered. If yes, the indoor air conditioner will be turned on.

Figure 3-18 Calling the message query API

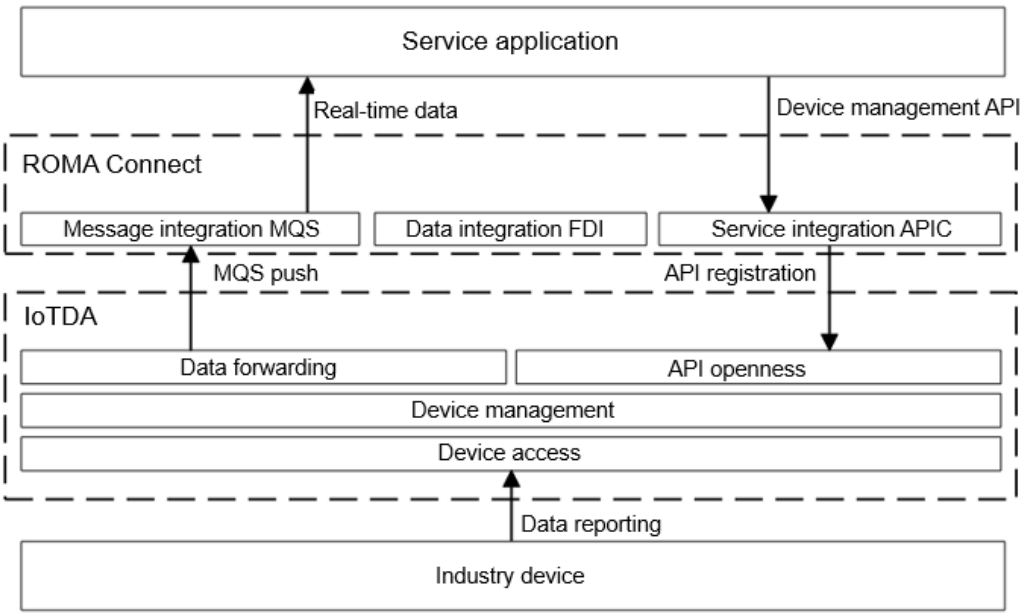


----End

3.6 Quick Integration with ROMA Connect

Scenarios

IoTDA provides unified device access in industry solutions, such as smart city and smart campus. However, these solutions also require comprehensive service capabilities such as video surveillance, meeting system, geographic information system (GIS), and data lake. To connect these systems to IoTDA, you need to integrate IoTDA with the unified data integration platform ROMA Connect. The following figure shows the common architecture for the integration of IoTDA and ROMA Connect.



IoTDA and ROMA Connect are integrated using the following to methods:

- APIs

  All northbound IoTDA APIs can be connected to ROMA APIC for upper-layer applications to call. For details about supported APIs, see API Reference.

- MQS push

  All real-time events and data of IoTDA can be pushed to ROMA MQS for unified subscription by upper-layer applications. For details about the supported real-time events and data types, see **Data Forwarding**.

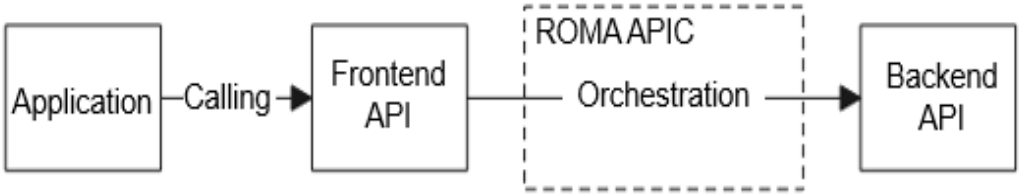
Prerequisites

Complete the following tasks to ensure successful integration of IoTDA and ROMA Connect.

1. You have an available ROMA Connect instance.
2. A physical or simulated device is connected to IoTDA and can report data. For details, see **Connecting MQTT Devices**.
3. You have created an access key (AK/SK, Access Key ID/Secret Access Key) for ROMA APIC authentication. For details about how to create an access key, see Access Key.

Configuring API Integration

As the backend service of ROMA APIC, IoTDA provides backend APIs for registration with APIC. Then, APIC encapsulates and orchestrates the APIs into frontend APIs for applications to call, as shown in the following figure.



The following uses the device query API as an example to describe how to register the API with ROMA APIC.

Parameters **project_id** and **device_id** of the backend device query API need to be transferred externally. The path parameter of the two backend APIs is mapped to the query parameter of the frontend API and exposed as the **GET /iot/devices** API.

Frontend API:

Request Method	GET
URI	/iot/devices?project_id={project_id}&device_id={device_id}
Transport Protocol	HTTPS

Backend API:

Request Method	GET
URI	/v5/iot/{project_id}/devices/{device_id}
Transport Protocol	HTTPS

- Creating a signature key

All IoTDA APIs at the application side require authentication. The following describes how to configure the access key of IoTDA on ROMA Connect.

Step 1 Log in to the ROMA Connect console and choose **API Connect** > **API Management**. On the **Signature Keys** tab page, click **Create**.

Step 2 Set **Type** to **hmac**. Enter the prepared AK and SK in **Key** and **Secret**, respectively. Click **OK**.

----End

- Creating an API

The following demonstrates how to use APIC to integrate the frontend device query API with IoTDA.

Step 1 Log in to the ROMA Connect console, choose **API Connect** > **API Management**, and click **Create API**.

Step 2 On the **Create API** page, set the API name to **QueryDeviceDetails**.

Step 3 Generally, there is no group or integration application for a new ROMA Connect instance. Click the marked links to create a group and integration application as prompted.

Step 4 Create the group and integration.

Step 5 See the ROMA Connect official documentation for details about the remaining settings. In this example, default settings are used.

Step 6 Click **Next** to configure the API request. Enter the frontend API for querying device details in the request path.

Step 7 Click **Add Input Parameter** to add the two query parameters **project_id** and **device_id** to **Input Parameters**.

Step 8 Click **Next**.

Step 9 In **Backend Address**, enter the northbound IoT application address. For details about how to obtain the address, see "Appendix" > "Obtaining the Application Access Address" in *API Reference*. In **Path**, enter the actual API path.

Step 10 APIC allows frontend API parameters to be transparently transmitted to the backend. You can click **Import Input Parameter** to import the input parameters

configured in the previous step. You can specify the parameter mapping in the backend parameter name. Parameters of the backend device query API are in **PATH**.

Step 11 Click **Next**, and then click **Publish API**.

Step 12 On the **Publish API** page, click **Publish**.

----End

- Binding a signature key to an API

Step 1 On the API details page, click **Signature Keys**.

Step 2 Click **Bind**.

Step 3 Select the created key and click **OK**.

----End

Verifying the API Integration

Step 1 Click **Debug** on the right part of the API details page.

Step 2 Enter values of **project_id** and **device_id**, and click **Send Request**.

Step 3 View the response displayed on the right.

----End

Configuring MQS Push

The following uses real-time device data reporting as an example to describe how to push data to ROMA MQS through data forwarding.

- Creating a topic

Step 1 Log in to the ROMA Connect console, choose **Message Queue Service** > **Topic Management**, and click **Create Topic**.

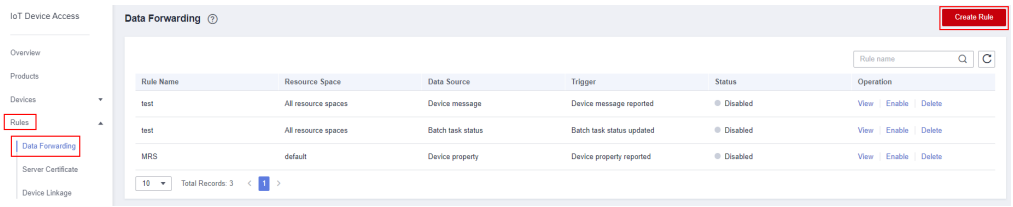
Step 2 Set **Topic Name** to **IOT_TEST** and set other parameters based on the site requirements. In this example, the default values are used. Click **OK**.

----End

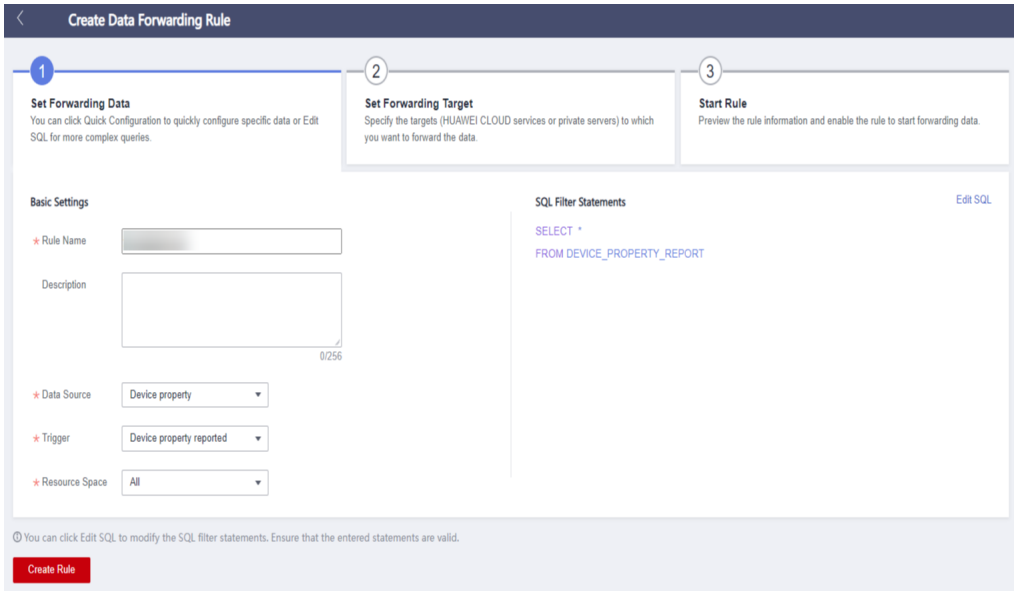
- Configuring data forwarding to MQS on IoTDA

Step 1 View the MQS connection address and authentication information on the **Instance Information** page of the ROMA Connect console.

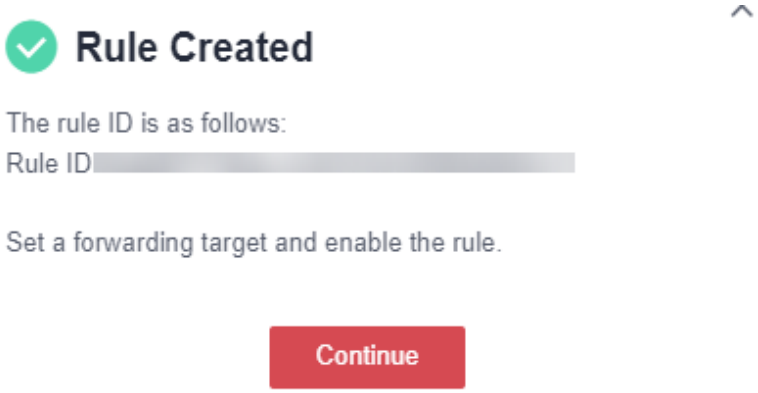
Step 2 On the IoTDA console, choose **Rules** > **Data Forwarding** > **Create Rule**.



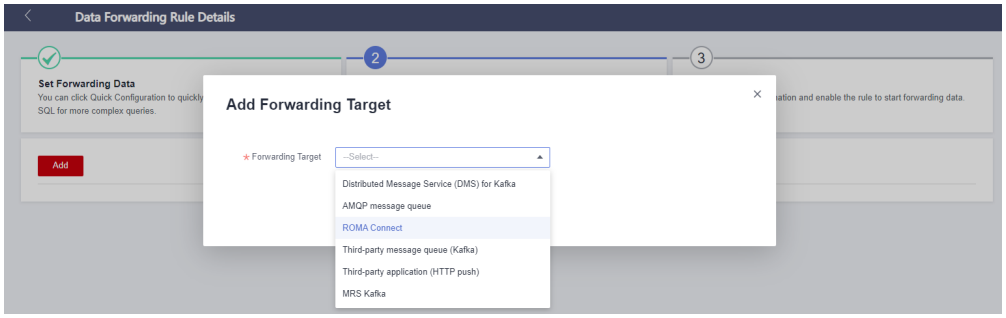
Step 3 Create a forwarding rule, as shown in the following figure.



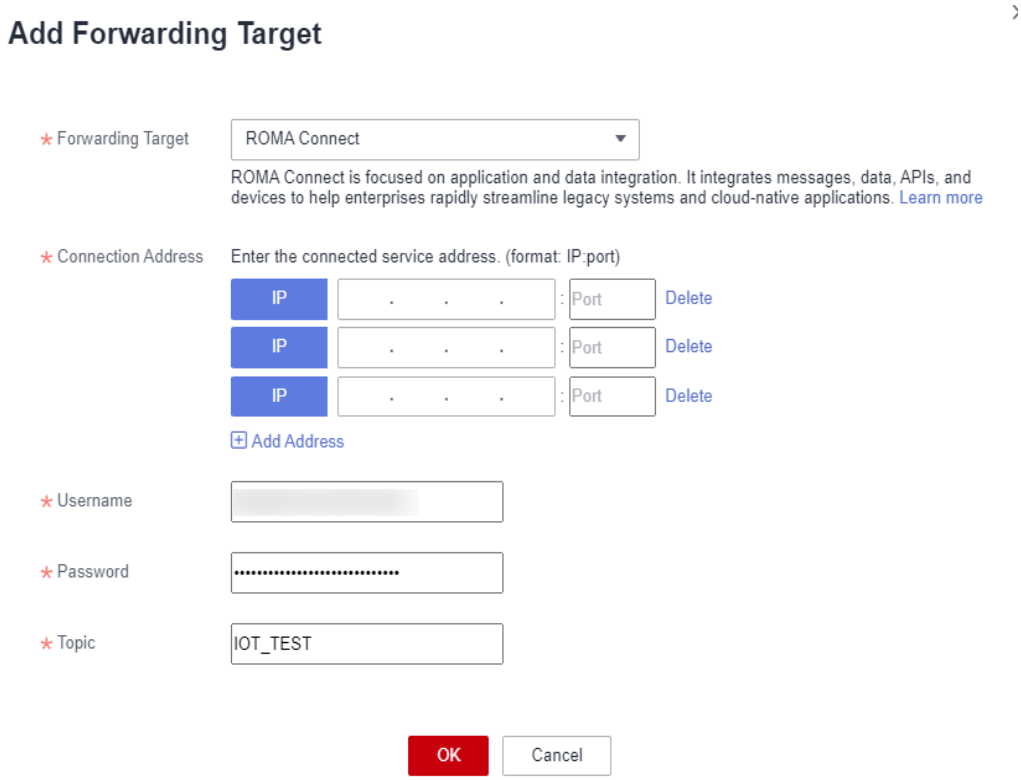
Step 4 Click **Continue** to set the forwarding target.



Step 5 Click **Add** and choose **ROMA Connect**.

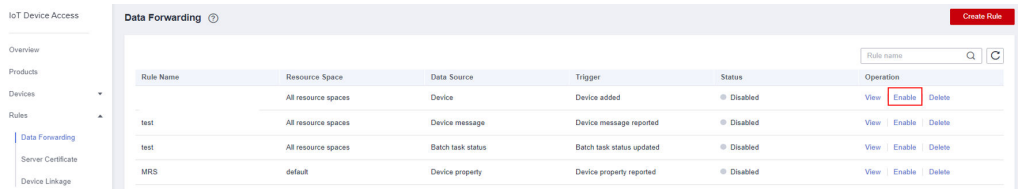


Step 6 Enter the MQS push address, username (AppKey), password (AppSecret), and topic, as shown in the following figure. (If there are multiple IP addresses and ports, click **Add Address** to add them.)



Step 7 Click **OK**.

Step 8 Return to the data forwarding page and click **Enable** to make the rule take effect.



----End

Verifying MQS Push

Step 1 Log in to the ROMA Connect console, choose **Message Queue Service** > **Message Query**. In the upper right corner, select **IOT_TEST**.

Step 2 If the real-time data reported by the device is displayed, the push is successful.

----End

3.7 Developing a Protocol Conversion Gateway for Access of Generic-Protocol Devices

Scenarios

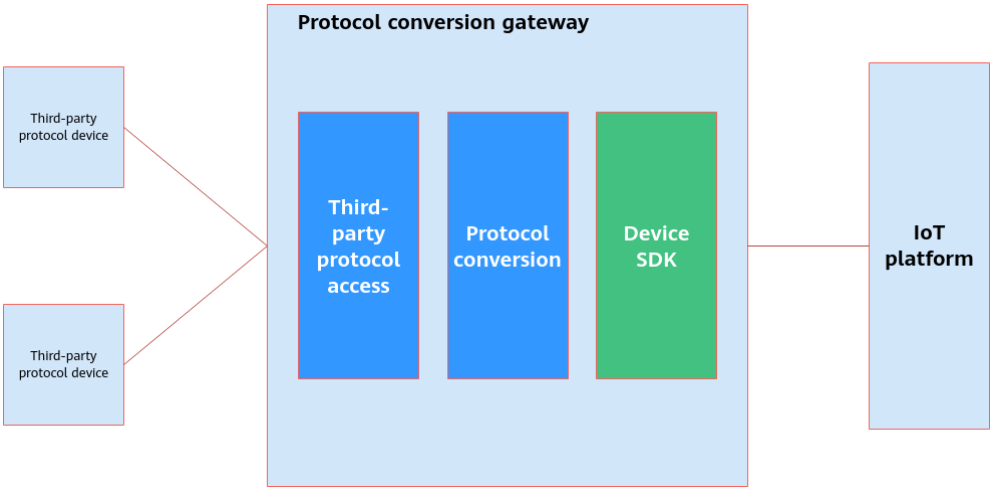
Currently, the platform supports only standard protocols such as MQTT, HTTP, and LwM2M. If a device uses other protocols (referred to as third-party protocols), it cannot access the platform directly.

To address this issue, protocol conversion must be performed outside the platform. It is recommended that a gateway be used to convert third-party protocols into MQTT. This gateway is called the protocol conversion gateway.

Principles

The figure below shows the overall architecture of the solution.

Figure 3-19 Protocol conversion gateway



The protocol conversion gateway can be deployed in the cloud or locally. A third-party protocol device is connected to the platform as a child device of the protocol conversion gateway.

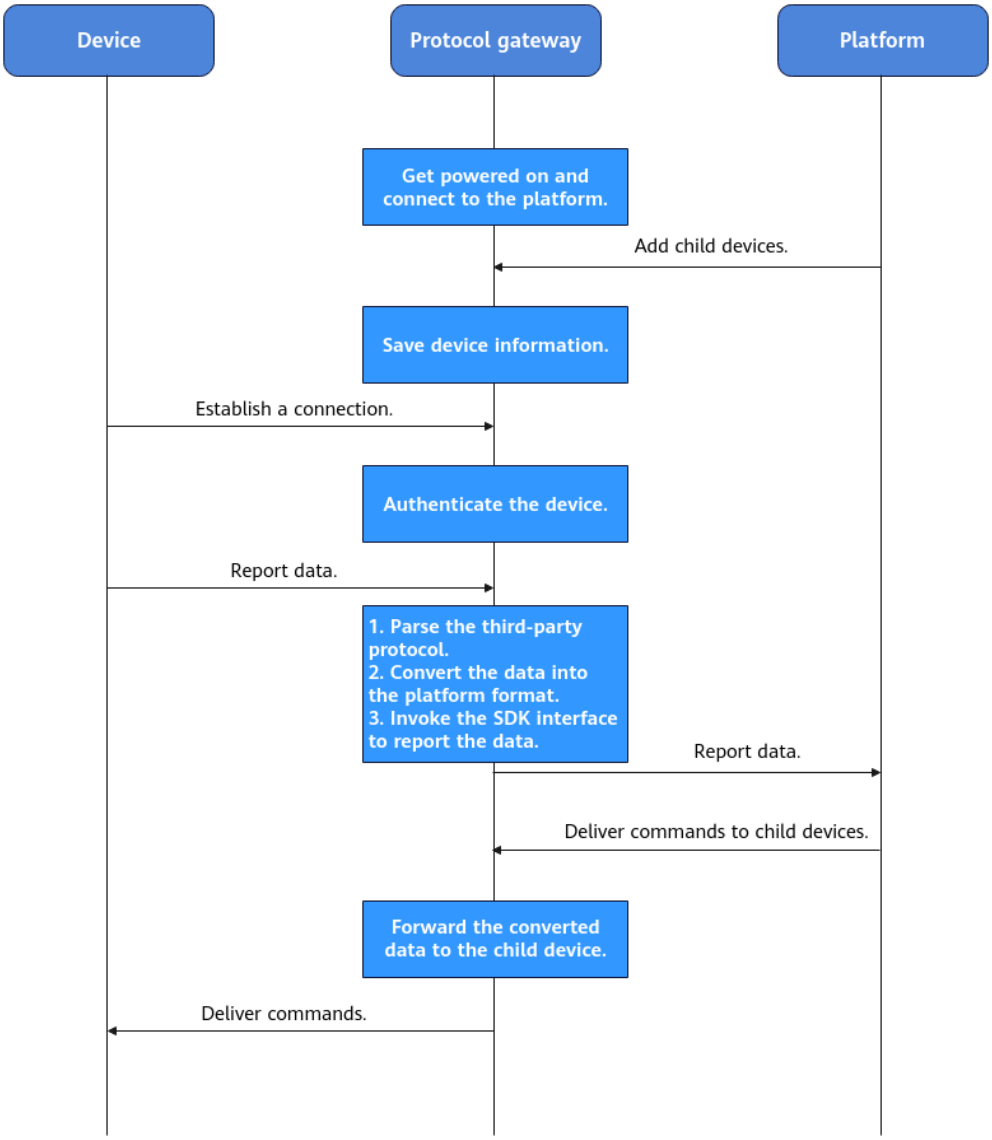
The protocol conversion gateway consists of the following components:

1. Third-party protocol access: This component parses and verifies third-party protocols.
2. Protocol conversion: This component converts between third-party protocol data and platform data.
   - In the upstream direction, the component converts third-party protocol data into platform-supported data and calls the device SDK APIs to report the data.
   - In the downstream direction, the component, after receiving data from the platform, converts the data into third-party protocol data and forwards the data to third-party protocol devices.

3. Device SDK: The component is the device access SDK provided by the platform and offers common gateway functions. You can implement your own gateways based on the SDK.

Service Flow

Figure 3-20 Service flow



1. Register a gateway on the IoT platform. For details, see "Devices" > "Registering a Device" in the *User Guide*.
2. Power on the gateway and connect it to the platform. Obtain the authentication parameters required for the connection from **1**.
3. Register a child device on the platform. The platform delivers a child device addition event to the gateway. The gateway saves the child device information persistently. (The SDK provides the default persistent implementation. You can customize the implementation.)
4. The child device connects to the gateway, and the gateway authenticates the child device.

5. The child device reports data to the gateway. The gateway converts the data to the format supported by the platform and then calls an SDK API for reporting child device properties or messages to the platform.
6. The platform delivers a command to the child device. The gateway converts the command into a command compliant with the third-party protocol and forwards it to the child device. The child device processes the command.

Implementation of Protocol Conversion Gateway

For details, see "Development on the Device Side" > "Access Using IoT Device SDKs" in the *Developer Guide*.

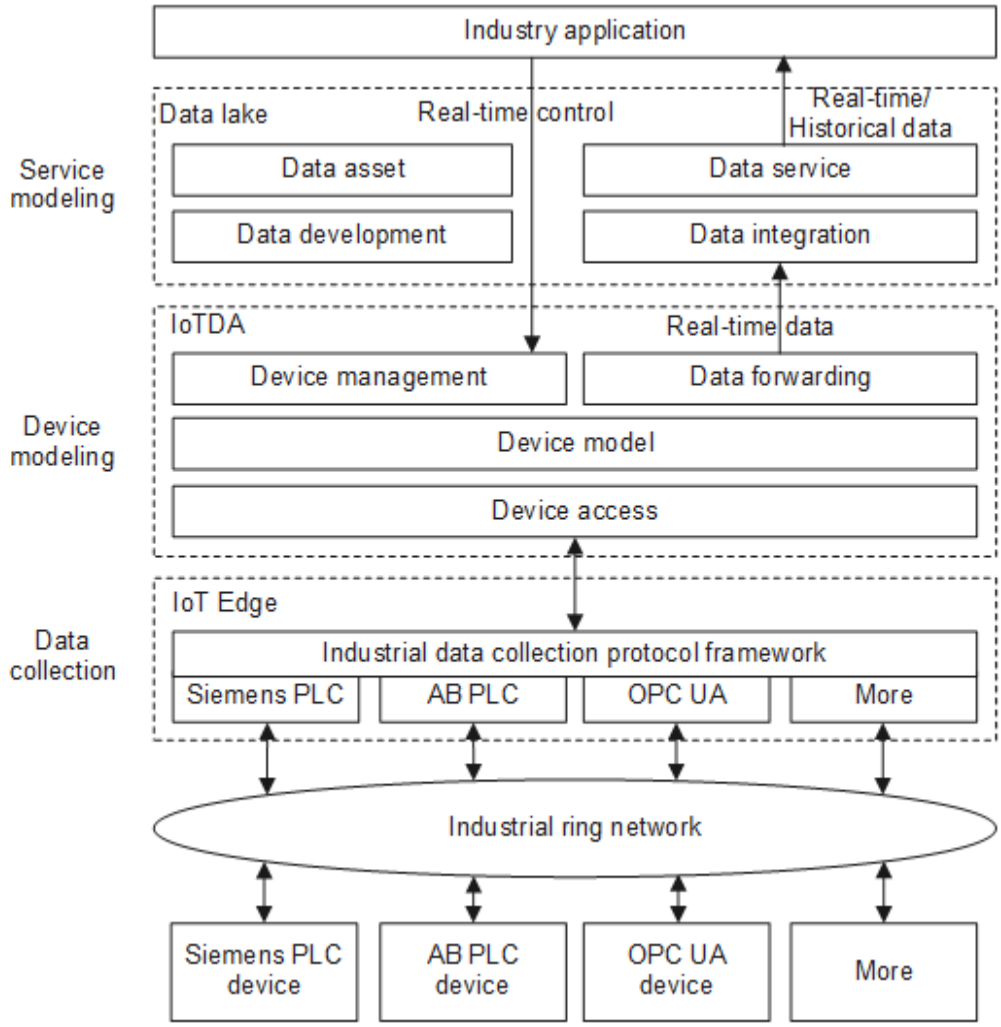
3.8 IoTDA in Industrial Data Collection

Scenarios

IoTDA is a general IoT platform that provides general capabilities, such as device connection, bidirectional communications between devices and the cloud, batch device management, and remote control and monitoring. Solutions can be customized based on project requirements to implement southbound and northbound connection and support scenario-specific applications at the upper layer. Industrial data collection is a typical use case of IoT technologies. IoTDA and IoTEdge gain expertise in industrial data collection from the practices in the coal mine, steel, automated machine, and food processing industries.

The following describes an industry application built based on industrial data collection and demonstrates how to transfer data from end to end using IoTDA and IoTEdge. The following figure shows the solution architecture.

Figure 3-21 Architecture of the industrial data collector solution



Architecture Overview

NOTE

Figure 3-21 shows a complete industrial data collection solution. Not all services are implemented by the platform. They are usually built together with customers or industry partners.

In this example, the platform implements data collection and device modeling. This topic focuses on the services provided by IoTDA at the device modeling layer and describes how IoTDA works with IoTEdge. IoTEdge implements the services at the data collection layer.

The architecture consists of data collection, device modeling, and service modeling layers from bottom to top. Core service functions of each layer are as follows:

- **Data collection layer:** In the industrial data collection solution, IoTEdge collects OT data. It integrates diverse industrial IoT protocols using the industrial data collection framework at the edge. This enables IoTEdge to read point data of the subsystem upper computer or bottom-layer programmable logic controller (PLC) and map the points to logical device models, thereby reporting the data to IoTDA.
- **Device modeling layer:** IoTDA aggregates and parses data collected by IoTEdge into data in the standard format, and pushes the data to the service

modeling layer based on forwarding rules. It also exposes unified device management APIs to the industry application.

- **Service modeling layer:** Data from the device modeling layer is the data of a logical device. However, there are parent-child relationships between logical devices (such as water pumps, sensors, and air valves) in industrial scenarios. For example, a compressor consists of an air valve, an air inlet pipe, and an air tank. Service modeling processes the parent-child relationship to present data in the granularity of assets to the upper layer.

At the industry application layer, you can quickly build applications using the real-time and historical data APIs exposed by the service modeling layer and the device control APIs exposed by the device modeling layer, without handling complex protocols and data formats of devices.

There are two main northbound data flows from IoTDA in the industrial data collection scenario, as shown in the following figure.

- **Real-time data**: You can push aggregated real-time data streams to the upper layer in a unified device model and format based on specific rules and granularities. For example, you can push parameters such as the motor voltage, current, and power of a factory to be monitored in real time. For details about the real-time data push, see **Data Forwarding**.
- **Real-time control**: IoTDA provides unified control APIs in the reverse control scenario based on the unified model built for devices at the bottom layer. For example, you can use the APIs to start or stop motors in a factory, turn on or off a circuit breaker in a substation, and move robotic arms.

The following describes how to create a device model and how to connect to upper-layer applications based on real-time data and real-time control.

Device Modeling Modes

A device model describes a device using a standard computer language. It defines the data and control commands supported by the device. This helps upper-layer applications understand diverse devices without understanding complex underlying protocols and different data formats.

An ideal device model must meet the following requirements:

- **Appropriate granularity**: For the modeling of a power supply system, the entire power supply system, substation, transformer, or even a measurement and control device on a transformer can be used as a model. In this example, the granularity of the model is reduced because a device model can be adjusted flexibly using a computer language. If the granularity is too large, reported data will be too large and inflexible and the model will deliver weak universality. If the granularity is too small, an excessive number of models, heavy workload, and complex logic between models will be generated. Based on the practical experience of power supply system modeling, it is appropriate to build each transformer as a model.
- **Universality**: A device model not only describes a device but also summarizes a type of devices. In project practices, a type of devices that share a model can be easily understood by the upper layer. This also helps you accumulate data assets to develop big data and AI applications.

NOTE

Device models are referred to as products in IoTDA. For details about products, see **Products**.

The following uses motors in the belt transportation system as an example to describe how to fill in modeling fields.

Service (Mandatory) (service_id)	Property/Parameter Name (Mandatory) (property_name)	Data Type (Mandatory) (data_type)	Operation (method)	Description (description)
real_time	Electric_Machinery_Speed	decimal	RW	Motor rotation speed
real_time	Electric_Machinery_Ta	decimal	R	Motor stator temperature (Ta)
real_time	Electric_Machinery_Tb	decimal	R	Motor stator temperature (Tb)
real_time	Electric_Machinery_Tc	decimal	R	Motor stator temperature (Tc)
real_time	Electric_Machinery_FrontT	decimal	R	Temperature of the motor front end
real_time	Electric_Machinery_AfterT	decimal	R	Temperature of the motor back end
real_time	Electric_Machinery_Ta_Upper	decimal	R	Motor stator temperature upper limit (Ta)
real_time	Electric_Machinery_Tb_Upper	decimal	R	Motor stator temperature upper limit (Tb)
real_time	Electric_Machinery_Tc_Upper	decimal	R	Motor stator temperature upper limit (Tc)
real_time	Electric_Machinery_FrontT_Upper	decimal	R	Temperature upper limit of the front end
real_time	Electric_Machinery_AfterT_Upper	decimal	R	Temperature upper limit of the back end
real_time	Electric_Machinery_Status	int	R	Motor operating status
real_time	Electric_Machinery_Fun1_Status	int	R	Fan operating status of motor 1

Service (Mandatory) (service_id)	Property/Parameter Name (Mandatory) (property_name)	Data Type (Mandatory) (data_type)	Operation (method)	Description (description)
real_time	Electric_Machinery_Fun2_Status	int	R	Fan operating status of motor 2
alarm	Electric_Machinery_Overload	int	R	Motor overload alarm signal
alarm	Electric_Machinery_FrontT_Alarm	int	R	Motor front axle temperature alarm signal
alarm	Electric_Machinery_AfterT_Alarm	int	R	Motor rear axle temperature alarm signal
alarm	Electric_Machinery_Ta_Alarm	int	R	Temperature alarm of motor stator 1
alarm	Electric_Machinery_Tb_Alarm	int	R	Temperature alarm of motor stator 2
alarm	Electric_Machinery_Tc_Alarm	int	R	Temperature alarm of motor stator 3

The preceding model is based on the actual motor working scenario. Motor data collected from the PLC is classified into real-time data (**real_time**) and alarm data (**alarm**). **real_time** carries the real-time data of working motors. **alarm** carries the alarm data of motors. The reporting frequency and priority of real-time data are different from those of alarm data in practice. Therefore, you are advised to distinguish them.

The key fields of the model are described as follows:

- Service (**service_id**): a set of capabilities of devices, such as **real_time** and **alarm**. You can define and adjust the field based on the site requirements.
- Property (**property_name**): name of the device property, which identifies the readable or writable points on the device. You are advised to set this field to an identifying name.
- Data type (**data_type**): data type of the device property. The value can be **int**, **decimal**, or **string**.
- Operation (**method**): operation type supported by a property. The value can be **R** (read-only) or **RW** (read-write).
- Description (**description**): property description. You are advised to set this field to the device data point name.

Interconnecting with Real-time Data

In the industrial data collection solution, IoTDA aggregates data from the IoTEdge data collection layer. The following describes how to forward the data to the upper layer.

Generally, a complete production line consists of different automation systems of multiple processes and auxiliary monitoring systems (such as environment and power monitoring). Taking the coal mining industry as an example, safety is always the top concern for the high-risk industry. From manual and semi-mechanized mining to mechanized and intelligent mining, the coal mining industry has improved the automation level and reduced underground personnel, and is moving towards the ultimate goal of intelligent unmanned mining. The mining and safety system that covers mining, tunneling, electromechanical equipment, transportation and ventilation therefore came into being.

Based on the coal mine data collection practices, the real-time data of mining, tunneling, electromechanical equipment, transportation and ventilation systems is the data foundation for intelligent mining. Data of each system needs to be collected from the bottom layer and pushed to the upper layer by system. Therefore, data aggregated on IoTDA must be distinguished by system.

The following describes the implementation solution. The format of each to-be-forwarded device data record cached on the platform is as follows:

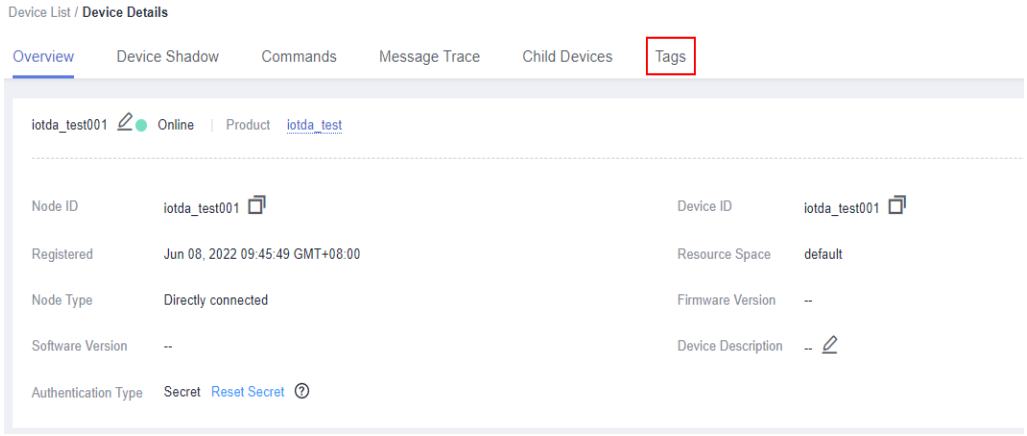
```
{
   "resource": "device.property",
   "event": "report",
   "event_time": "20151212T121212Z",
   "notify_data": {
      "header": {
         "device_id": "d4922d8a-6c8e-4396-852c-164aefa6638f", // Device ID.
         "product_id": "ABC123456789",  // ID of the product to which the device belongs.
         "app_id": "d4922d8a-6c8e-4396-852c-164aefa6638f",  // Resource space ID.
         "gateway_id": "d4922d8a-6c8e-4396-852c-164aefa6638f", // The gateway ID is displayed if the device is a child device.
         "node_id": "ABC123456789",  // Node ID of the device.
         "tags": [  // Key-value pairs customized on the device details page for the push after SQL filtering.
            {
               "tag_value": "topic",
               "tag_key": "ZCXT"
            }
         ]
      },
      "body": {   // Property data reported by the device in the format defined in the device model.
         "services": [
            {
               "service_id": "real_time",
               "properties": {
                  "CMJYXFX": 0,
                  "CMJYXSD": 0,
                  "DQCG_Z": 0.0,
                  "DQCG_Y": 0.0,
                  "CMJXDZJWZ": 0
               },
               "event_time": "20151212T121212100Z"
            }
         ]
      }
   }
}
```


The pushed data is a standard JSON character string. You can filter the JSON data by key value using the SQL syntax. For details about the supported SQL filtering functions, see **SQL Statements**.

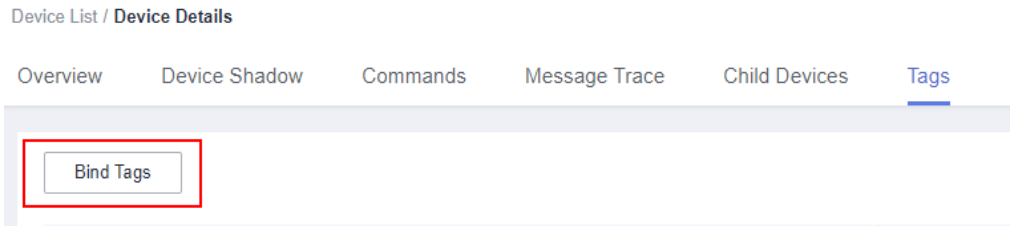
The **notify_data.header.tags** field is reserved for you to customize tags for data forwarding. Data can be filtered and forwarded based on the SQL syntax. If you tag each message with the system to which it belongs, data can be filtered by system based on tags during SQL-based forwarding.

The following describes how to configure a forwarding rule based on the fully-mechanized mining system.

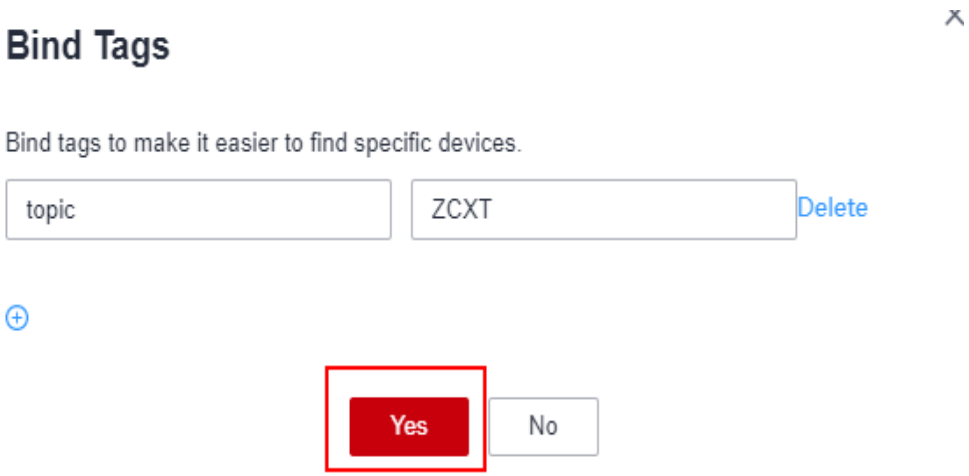
Step 1 Select the device to be tagged with its system to access the device details page.



Step 2 Click the **Tags** tab.

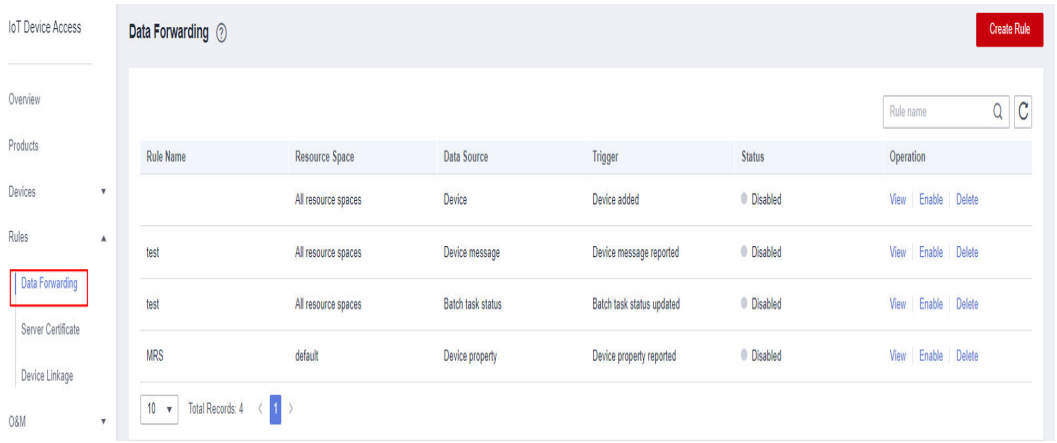


Step 3 Add a tag of system information, for example, **topic:ZCXT**. **ZCXT** indicates the fully-mechanized mining system. **topic** indicates the tag name, which will be used in SQL-based forwarding.



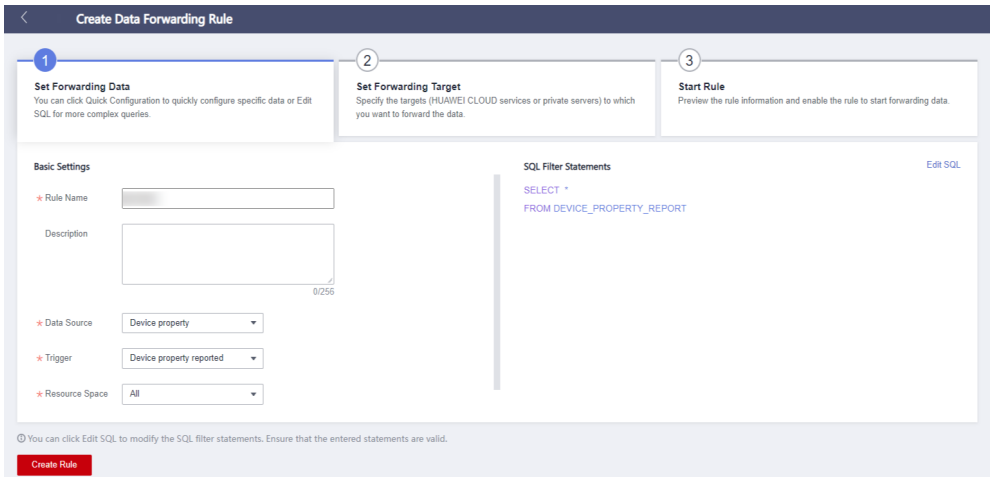
Step 4 Click **Yes**.

Step 5 Choose **Rules** > **Data Forwarding** in the navigation pane.

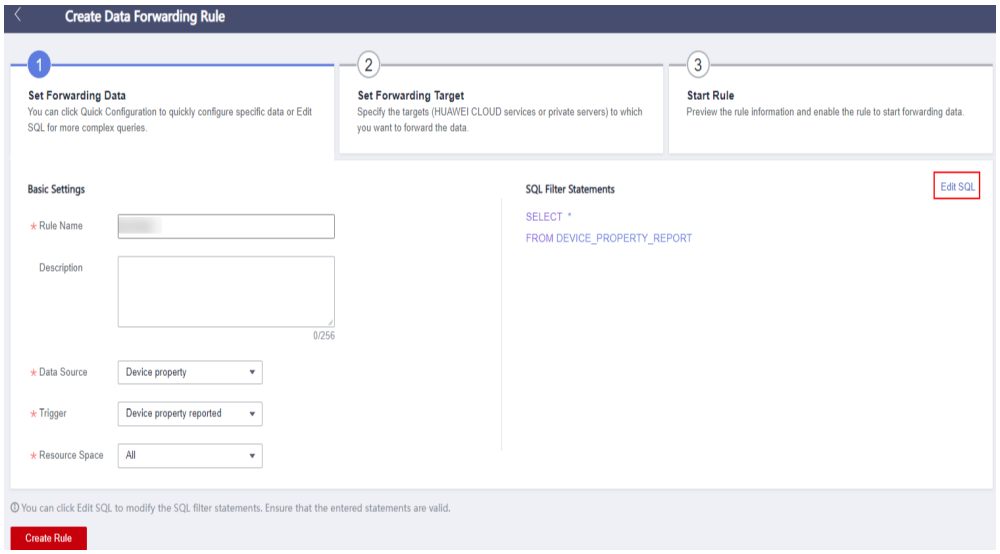


Step 6 Click **Create Rule** in the upper right corner to configure a rule to forward data by system.

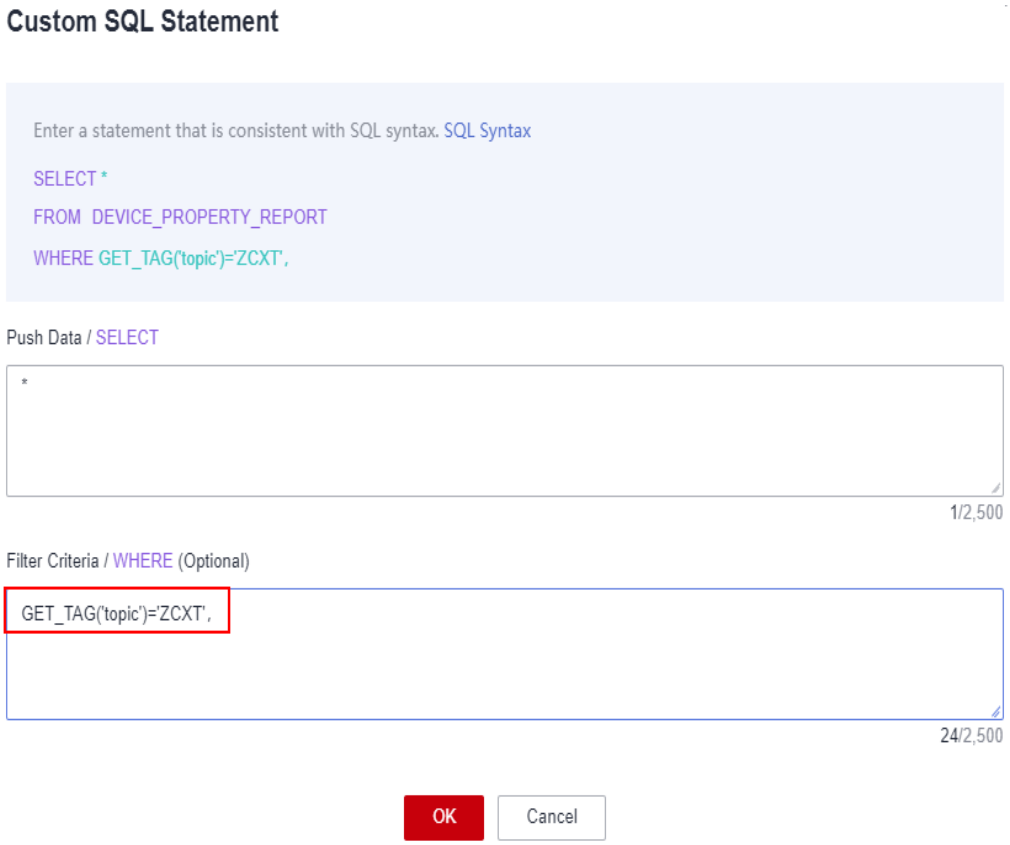
Step 7 Enter the rule name **Fully-mechanized mining system**.



Step 8 Click **Edit SQL** on the right and enter a custom SQL statement.



Step 9 In **Filter Criteria**, enter **GET_TAG('topic')='ZCXT'** to obtain the **topic** tag value. If the value is **ZCXT**, the data will be forwarded.



Step 10 Click **OK**.

Step 11 Configure the forwarding target and enable the rule. For details, see **Data Forwarding**.

----End

Implementing Real-time Control

In the industrial data collection scenario, the differences between the two APIs are as follows:

- **Modifying device properties:** This API is used to modify and write readable and writable properties in the device model. An HTTP request is responded only after the modification is successful or fails. Generally, point write is complete upon HTTP response completion. This API is recommended for the industrial data collection scenario.
- **Configuring desired properties in a device shadow**: A device shadow stores the desired value of the device property. The platform periodically compares the reported value with the desired value in the background. If the values are different, the platform keeps delivering the desired value to the device. The HTTP response completion indicates that the shadow configuration is complete. It is applicable only to store device configuration data.

4 FAQs

4.1 Product Models

4.1.1 How Can I Develop a Product Model?

IoTDA supports online and offline product model development. For details, see "Product Development" > "Developing a Product Model" in the *Developer Guide*.

4.2 Data Reporting

4.2.1 How Do I Handle Data Reporting Failure?

1. If the device that attempted to report data has been deleted from IoTDA, re-register the device and have the device report data again.
2. Check whether the product information specified when you called an API to register the device is consistent with that in the product model.
3. Check whether the property names in the reported message are the same as the service properties defined in the product model.
4. If the fault persists, check whether the network connection between the device and IoTDA is normal and whether the device is running properly.

4.2.2 Why Does a Device Report Data Successfully at One Location but Fail Elsewhere?

Contact the NB-IoT network carrier to check whether the NB-IoT card has geographical restrictions and check the local NB-IoT network status.

4.3 Command Delivery

4.3.1 How Do I Handle Command Delivery Failure?

An error occurs when you call the API for delivering a command, or the API call succeeded but the device does not receive the command. The possible causes are as follows:

- The device is offline.
- The device has not subscribed to the topic to which the command is published.
- The command delivery times out.

The following sections describe how to rectify the fault.

Device Offline

If the device is offline and you call the API for delivering the command, the error code **IoTDA.014016** is returned.

On the IoTDA console, choose **Devices** > **All Devices**, search for the device, and view the device status.

If the device status is **Offline**, connect the device first. If the device status is **Inactive**, initialize the device and connect the device first.

No Subscription to the Topic

Check whether the device has subscribed to the correct downstream topic. (The API for delivering a command can be called only after the subscription is successful. If the subscription fails, the device cannot receive messages from IoTDA.)

```
$oc/devices/{device_id}/sys/commands/#
```

{device_id} indicates the device ID.

Command Delivery Timeout

After IoTDA sends a command to a device, it waits for a response from the device. If the device does not respond to the command, a message is displayed indicating that the delivery timed out. Enable the device to report a response as soon as it receives the command. For details about the data format of the response, see "API Reference on the Device Side" > "Device Command API" > "Upstream Response Parameters" in the *API Reference*.

4.3.2 How Do I Deliver Commands to a CoAP Device?

Log in to the IoTDA console, and choose **Devices** > **All Devices**. Click **View** in the row where the device is located, click the **Commands** tab, and deliver a command in the **Asynchronous Command Delivery** area.

For details, see "User Guide" > "Message Communications" > "Command Delivery" > "Command Delivery for Devices Using CoAP" in the *Usage Guide*.

4.3.3 How Do I Deliver Commands to an MQTT Device?

Log in to the IoTDA console, and choose **Devices** > **All Devices**. Click **View** in the row where the device is located, click the **Commands** tab, and deliver a command in the **Synchronous Command Delivery** area.

For details, see "Message Communications" > "Command Delivery" > "Command, Property, and Message Delivery for MQTT Devices" in the *User Guide*.

4.4 Software or Firmware Upgrade

4.4.1 What Is a Software or Firmware Upgrade?

A software upgrade refers to the upgrade of the system software and application software of the device. A firmware upgrade refers to the upgrade of the underlying drivers of the device hardware.

To upgrade the software or firmware, upload the software or firmware package to IoTDA. Devices can obtain the software or firmware package from IoTDA for remote upgrade.

4.4.2 Can I Download Software or Firmware Packages from Third-Party Servers to IoTDA?

No. However, you can upload such packages to IoTDA. To upload a software or firmware package, log in to the IoTDA console, choose **Devices** > **Software/Firmware Upgrades**, click the **Software List** or **Firmware List** tab under **Manage Resource Package**, and click **Upload**.

4.5 Edge Devices

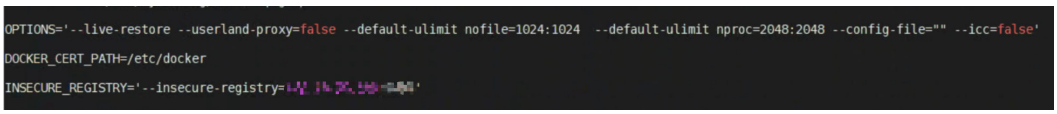
4.5.1 Why Does an Edge Device Fail MQTT Authentication?

When creating an MQTT device under an edge node, do not set the module ID parameter. Otherwise, MQTT authentication fails.

4.5.2 Why Cannot Docker Bridges Communicate with Each Other After the OS of Atlas 500 Is Upgraded to 22.0?

Security rules are added to new versions of EulerOS. Startup parameters are added to Docker startup items. If Docker bridges need to communicate with each other, perform the following operations:

Step 1 vi /etc/sysconfig/docker



Delete **--userland-proxy=false**, **--icc=false**, and **--config-file=""** from **OPTIONS** and save it. The result is as follows:

```
OPTIONS='--live-restore --default-ulimit nofile=1024:1024 --default-ulimit nproc=2048:2048'
```

Step 2 Restart Docker.

```
systemctl daemon-reload
systemctl restart docker
```

----End

4.5.3 Why Is There No Password Text Box When I Change the Password of an Edge Device?

If there is no text box for entering the password when you change the password of an existing MQTT device under an edge node, the desired password value in the device shadow may be canceled on the IoTDA console.

Log in to the IoTDA console. On the **Device Details** page, click the **Device Shadow** tab, and click **Configure Property** to configure a desired value for the **password** property of the MQTT device whose service ID is **$config**. You can set the **password** property to any value. Return to the edge node details page and configure a password for the MQTT device.

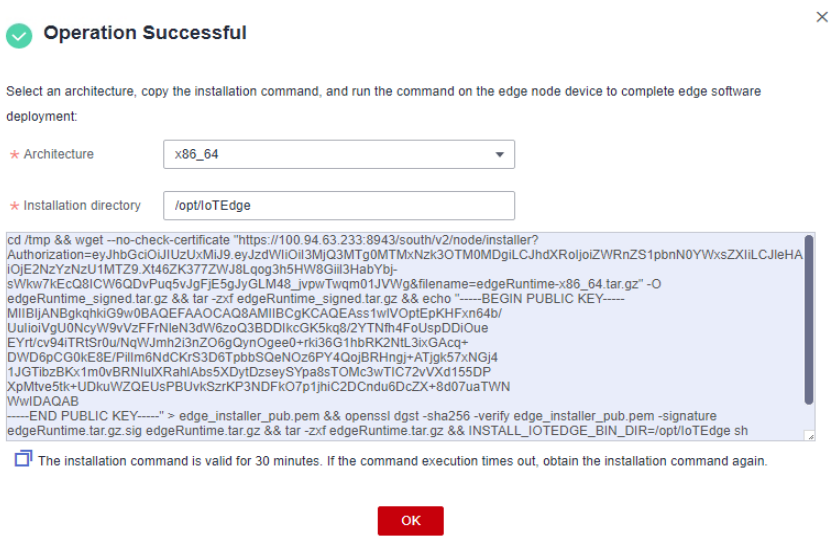
4.6 Clock Synchronization on Edge Nodes

This section describes how to configure clock synchronization on edge nodes.

4.6.1 Configuration File Directory

The default configuration file directory is **${EDGE_DAEMON_INSTALL_DIR}/IoTEdge/edgeDaemon/conf/defaultConfig.json**.

EDGE_DAEMON_INSTALL_DIR indicates the edgedaemon installation directory. Specify the directory when obtaining the installation command. The default value is **/opt/IoTEdge**. The following figure shows the details.



4.6.2 Configuration Example

The configuration items related to clock synchronization are as follows:

```
{
"skip_signature_verification": "false",
"clock_config": {
"enable_time_sync": "true",
"delay_threshold": "10",
"delay_threshold_ms": "20",
"calibration_period": "10",
"enable_sync_hwclock": "true"
}
}
```

4.6.3 Configuration Items

1. **skip_signature_verification**: Ignore this configuration item, which is irrelevant to clock synchronization.
2. **clock_config**: clock synchronization configurations.
3. **enable_time_sync**: whether to enable clock synchronization. **true** indicates that clock synchronization is enabled. **false** indicates that clock synchronization is disabled. The default value is **false**.
4. **delay_threshold**: maximum offset, that is, the threshold for clock calibration. If the threshold is reached, clock calibration is performed. The unit is second.
5. **delay_threshold_ms**: maximum offset, that is, the threshold for clock calibration. If the threshold is reached, clock calibration is performed. The unit is millisecond. The default value is **1000**. Note that when this parameter is set, **delay_threshold** is invalid, and the value of **delay_threshold_ms** prevails.
6. **calibration_period**: interval at which clock calibration is performed, in seconds. The minimum value is 15 seconds, and the default value is 1 hour.
7. **enable_sync_hwclock**: whether to update the hardware clock synchronously. After the restart, the hardware clock is used. If the hardware clock is inconsistent with the system clock, the system clock is reset to the same time as the hardware clock after the restart.

4.6.4 Precautions

After the configuration is modified, save the file and restart the edgedaemon process for the modification to take effect.

Note: Restarting the daemon process will restart all applications, so you need to perform configuration after the initial node installation.

Restart command:

```
systemctl restart edgedaemon
```

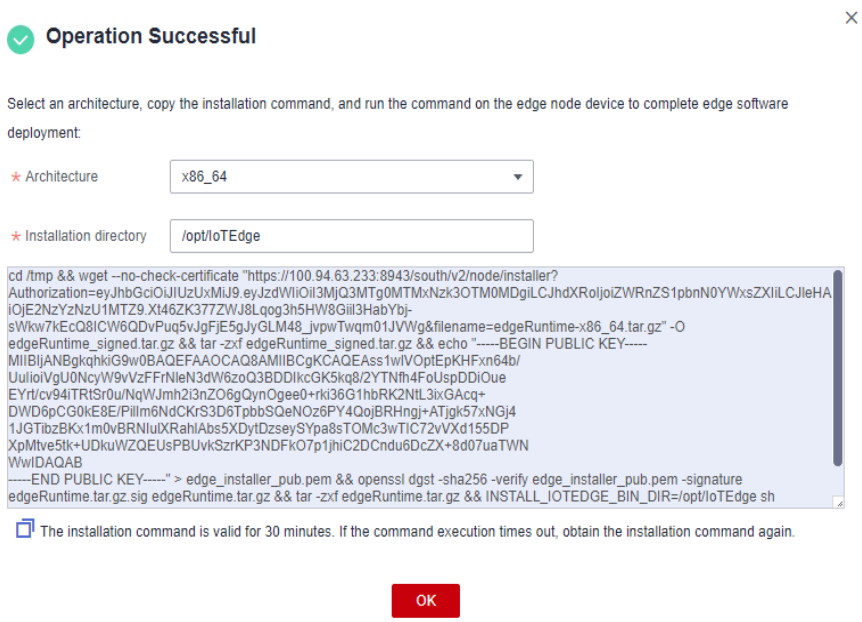
4.7 Edge Node Bridge Configuration

After an edge node is created, a bridge named **iot-edge-bridge** is created for network access of edge applications. If a network access conflict occurs, you can modify the network segment of the bridge to rectify the fault.

4.7.1 Configuration File Directory

The default configuration file directory is **${EDGE_DAEMON_INSTALL_DIR}/IoTEdge/edgeDaemon/conf/defaultConfig.json**.

EDGE_DAEMON_INSTALL_DIR indicates the edgedaemon installation directory. Specify the directory when obtaining the installation command. The default value is **/opt/IoTEdge**. The following figure shows the details.



4.7.2 Configuration Example

Example of edge node bridge configuration items:

```
{
"network_config": {
"ipv4_subnet": "169.254.254.0/24",
"ipv4_gateway": "169.254.254.1",
"ipv6_subnet": "fd0e::169:254:254:1/120",
"ipv6_gateway": "fd0e::169:254:254:1"
}
}
```

4.7.3 Configuration Items

1. **network_config**: network-related configuration of the self-built bridge **iot-edge-bridge**.
2. **ipv4_subnet**: the bridge network segment. The default value is **169.254.254.0/24**. In earlier versions, the value is **172.20.0.0/24**.
3. **ipv4_gateway**: default gateway of the bridge. The default value is **169.254.254.1**. In earlier versions, the value is **172.20.0.1**.
4. **ipv6_subnet**: IPv6 network segment of the bridge. The default value is **fd0e::169:254:254:1/120**.
5. **ipv6_gateway**: default IPv6 gateway of the bridge. The default value is **fd0e::169:254:254:1**.

4.7.4 Precautions

After the configuration is modified, save the file and restart the edgedaemon process for the modification to take effect.

Note: Restarting the daemon process will restart all applications, so you need to perform configuration after the initial node installation.

Restart command:

```
systemctl restart edgedaemon
```

