ALIBABA CLOUD

Alibaba Cloud

物联网平台

Best Practices

Document Version: 20220624

Alibaba Cloud

Legal disclaimer

Alibaba Cloud reminds you to carefully read and fully understand the terms and conditions of this legal disclaimer before you read or use this document. If you have read or used this document, it shall be deemed as your total acceptance of this legal disclaimer.

1. You shall download and obtain this document from the Alibaba Cloud website or other Alibaba Cloud-authorized channels, and use this document for your own legal business activities only. The content of this document is considered confidential information of Alibaba Cloud. You shall strictly abide by the confidentiality obligations. No part of this document shall be disclosed or provided to any third party for use without the prior written consent of Alibaba Cloud.

2. No part of this document shall be excerpted, translated, reproduced, transmitted, or disseminated by any organization, company or individual in any form or by any means without the prior written consent of Alibaba Cloud.

3. The content of this document may be changed because of product version upgrade, adjustment, or other reasons. Alibaba Cloud reserves the right to modify the content of this document without notice and an updated version of this document will be released through Alibaba Cloud-authorized channels from time to time. You should pay attention to the version changes of this document as they occur and download and obtain the most up-to-date version of this document from Alibaba Cloud-authorized channels.

4. This document serves only as a reference guide for your use of Alibaba Cloud products and services. Alibaba Cloud provides this document based on the "status quo", "being defective", and "existing functions" of its products and services. Alibaba Cloud makes every effort to provide relevant operational guidance based on existing technologies. However, Alibaba Cloud hereby makes a clear statement that it in no way guarantees the accuracy, integrity, applicability, and reliability of the content of this document, either explicitly or implicitly. Alibaba Cloud shall not take legal responsibility for any errors or lost profits incurred by any organization, company, or individual arising from download, use, or trust in this document. Alibaba Cloud shall not, under any circumstances, take responsibility for any indirect, consequential, punitive, contingent, special, or punitive damages, including lost profits arising from the use or trust in this document (even if Alibaba Cloud has been notified of the possibility of such a loss).

5. By law, all the contents in Alibaba Cloud documents, including but not limited to pictures, architecture design, page layout, and text description, are intellectual property of Alibaba Cloud and/or its affiliates. This intellectual property includes, but is not limited to, trademark rights, patent rights, copyrights, and trade secrets. No part of this document shall be used, modified, reproduced, publicly transmitted, changed, disseminated, distributed, or published without the prior written consent of Alibaba Cloud and/or its affiliates. The names owned by Alibaba Cloud shall not be used, published, or reproduced for marketing, advertising, promotion, or other purposes without the prior written consent of Alibaba Cloud. The names owned by Alibaba Cloud include, but are not limited to, "Alibaba Cloud", "Aliyun", "HiChina", and other brands of Alibaba Cloud and/or its affiliates, which appear separately or in combination, as well as the auxiliary signs and patterns of the preceding brands, or anything similar to the company names, trade names, trademarks, product or service names, domain names, patterns, logos, marks, signs, or special descriptions that third parties identify as Alibaba Cloud and/or its affiliates.

6. Please directly contact Alibaba Cloud for any errors of this document.

Document conventions

Style	Description	Example
Danger	A danger notice indicates a situation that will cause major system changes, faults, physical injuries, and other adverse results.	**Danger:** Resetting will result in the loss of user configuration data.
Warning	A warning notice indicates a situation that may cause major system changes, faults, physical injuries, and other adverse results.	**Warning:** Restarting will cause business interruption. About 10 minutes are required to restart an instance.
Notice	A caution notice indicates warning information, supplementary instructions, and other content that the user must understand.	**Notice:** If the weight is set to 0, the server no longer receives new requests.
Note	A note indicates supplemental instructions, best practices, tips, and other content.	**Note:** You can use Ctrl + A to select all files.
>	Closing angle brackets are used to indicate a multi-level menu cascade.	Click **Settings**> **Network**> **Set network type**.
Bold	Bold formatting is used for buttons , menus, page names, and other UI elements.	Click **OK**.
`Courier font`	Courier font is used for commands	Run the `cd /d C:/window` command to enter the Windows system folder.
Italic	Italic formatting is used for parameters and variables.	`bae log list --instanceid` *Instance_ID*
[] or [a|b]	This format is used for an optional value, where only one item can be selected.	`ipconfig [-all|-t]`
{} or {a|b}	This format is used for a required value, where only one item can be selected.	`switch {active|stand}`

Table of Contents

1.Device access 07
1.1. Access IoT Platform by using Paho 07
1.1.1. Use the Paho MQTT C client 07
1.1.2. Use the Paho MQTT Java client 13
1.1.3. Use the Paho MQTT Go client 17
1.1.4. Use the Paho MQTT C# client 22
1.1.5. Use the Paho MQTT Android client 26
1.1.6. Troubleshooting 31
1.2. Access IoT Platform by using CoAP 34
1.3. Access IoT Platform by using HTTP 41
1.4. Access IoT Platform by using MQTT over WebSocket conne... 45
1.5. MQTT-based dynamic registration 50
1.6. HTTPS-based dynamic registration 56
1.7. Connect a device to IoT Platform by using MQTT.fx 61
1.8. Connect Android Things hardware to IoT Platform 68
1.9. Connect an RTOS device to IoT Platform by using a TCP c... 73
1.9.1. Overview 73
1.9.2. Create a product and add a device 74
1.9.3. Build a development environment for devices 75
1.9.4. Develop for a device 80
1.10. Connect a bare-metal device to IoT Platform by using TC... 85
1.10.1. Overview 85
1.10.2. Create a product and add a device 86
1.10.3. Build a development environment for devices 87
1.10.4. Provision a device 91
1.11. Connect a serial device to IoT Platform by using a DTU d... 96

1.11.1. Overview 96
1.11.2. Configure a product and a device 97
1.11.3. Configure a DTU device 102
1.11.4. Test data communication 103
1.12. Connect environmental sensors to IoT Platform by using ... 104
1.13. Connect a Linux device to IoT Platform 110
2.Communications 114
2.1. Use custom topics for communication 114
2.2. Remotely control a Raspberry Pi server 119
2.3. Establish TSL-based communication 129
2.4. Communication between devices 141
2.4.1. M2M communication 141
2.4.2. Use the rules engine to establish M2M communication 142
2.4.3. Use topic-based message routing to establish M2M co... 144
2.5. Create an MNS server-side subscription 146
2.6. Parse pass-through data from devices 150
3.Device management 167
3.1. Set a desired property value to control the bulb status 167
3.2. Connect a sub-device to IoT Platform 181
3.2.1. Overview 181
3.2.2. Create a gateway and a sub-device 183
3.2.3. Initialize the SDKs 183
3.2.4. Connect the gateway device to IoT Platform 184
3.2.5. Connect the sub-device to IoT Platform 187
4.Maintenance and Monitoring 198
4.1. Use CloudMonitor to monitor IoT resources 198
4.1.1. Overview 198
4.1.2. Create a product and devices 199

4.1.3. Configure alert contacts 203
4.1.4. Create event-triggered alert rules 204
4.1.5. Create threshold-triggered alert rules 205
4.2. Configure OTA updates for devices 207
4.2.1. Overview 207
4.2.2. Configure OTA update for devices 209
4.2.3. Push firmware files to devices 217
5.Data forwarding 221
5.1. Use the Alibaba Cloud big data platform to create a dash... 221
5.2. Push device data to DingTalk groups 224

1.Device access

1.1. Access IoT Platform by using Paho

1.1.1. Use the Paho MQTT C client

This article describes how to use the open source Paho MQTT project for embedded C to connect and communicate with IoT Platform.

Prerequisites

A product and a device are created in the IoT Platform console.The device certificate information including ProductKey, DeviceName, and DeviceSecret is obtained. For more information, see the following articles:

- View the endpoint of an instance。
- Create a product
- Create a device

Prepare the development environment

In this example, Ubuntu 16.04-LTS is used to build the development environment. Run the following commands to build the development environment:

```
sudo apt-get update
sudo apt-get install build-essential git sed cmake
```

Download the Paho MQTT library for C

Run the following command to clone the Paho MQTT library for C:

```
git clone https://github.com/eclipse/paho.mqtt.embedded-c.git
```

> **Note** The master branch is used to develop the sample code in this article. The commit ID is `29ab2aa29c5e47794284376d7f8386cfd54c3eed`.

The Paho MQTT project for embedded C includes three sub-projects:

- *MQTTPacket*: provides serialization and deserialization of MQTT data packets and some helper functions.
- *MQTTClient*: encapsulates the high-level C++ client program generated by *MQTTPacket*.
- *MQTTClient-C*: encapsulates the high-level C client program generated by *MQTTPacket*.

  The *MQTTClient-C* project includes the following directories and files:

```
├── CMakeLists.txt
├── samples
│   ├── CMakeLists.txt
│   ├── FreeRTOS
│   └── linux
├── src
│   ├── CMakeLists.txt
│   ├── FreeRTOS
│   ├── MQTTClient.c
│   ├── MQTTClient.h
│   ├── cc3200
│   └── linux
└── test
    ├── CMakeLists.txt
    └── test1.c
```

- The *samples* directory provides sample code for *FreeRTOS* and *Linux* systems.
- The *src* directory provides the code for *MQTTClient* and network drivers for porting to FreeRTOS, CC3200, and Linux.

For more information about the Paho MQTT API, see *MQTTClient.h*.

Connect the client to IoT Platform

1. Click aiot_mqtt_sign.c and copy the source code that is provided by Alibaba Cloud to obtain the MQTT connection parameters. Then, create the *aiot_mqtt_sign.c* file on premises and paste the code to the file.

   The *aiot_mqtt_signal.c* file defines the aiotMqttSign() function.

   - Syntax

     ```
     int aiotMqttSign(const char *productKey, const char *deviceName, const char *deviceSe
     cret,
                      char clientId[150], char username[65], char password[65]);
     ```

   - Description

     This class is used to obtain the following MQTT connection parameters: username, password, and clientid.

   - Input parameters

Parameter	Type	Description
productKey	const char *	The ProductKey of the product to which the device belongs. This parameter is used to identify the device in IoT Platform.
deviceName	const char *	The DeviceName of the device. This parameter is used to identify the device in IoT Platform.
deviceSecret	const char *	The DeviceSecret of the device. This parameter is used to identify the device in IoT Platform.

   - Output parameters

Parameter	Type	Description
username	char *	The username that is used to establish the MQTT connection.
password	char *	The password that is used to establish the MQTT connection.
clientId	char *	The ID of the MQTT client.

- Response codes

Response code	Description
0	The function is implemented.
-1	The function failed to be implemented.

2. Add a program file that can connect a device to IoT Platform.

   You must write a program to call the aiotMqttSign() function in the *aiot_mqtt_sign.c* file to obtain the parameters that are used to establish an MQTT connection with IoT Platform.

   This section provides the development instructions and sample code.

   - Call the aiotMqttSign() function to obtain the clientId, username, and password parameters.

```
#define EXAMPLE_PRODUCT_KEY            "a11xsrW****"
#define EXAMPLE_DEVICE_NAME            "paho_****"
#define EXAMPLE_DEVICE_SECRET       "Y877Bgo8X5owd3lcB5wWDjryNPoB****"
extern int aiotMqttSign(const char *productKey, const char *deviceName, const char *d
eviceSecret,
                        char clientId[150], char username[65], char password[65]);
/* invoke aiotMqttSign to generate mqtt connect parameters */
char clientId[150] = {0};
char username[65] = {0};
char password[65] = {0};
if ((rc = aiotMqttSign(EXAMPLE_PRODUCT_KEY, EXAMPLE_DEVICE_NAME, EXAMPLE_DEVICE_SECRE
T, clientId, username, password) < 0)) {
    printf("aiotMqttSign -%0x4x\n", -rc);
    return -1;
}
printf("clientid: %s\n", clientId);
printf("username: %s\n", username);
printf("password: %s\n", password);
```

   - Connect the client to IoT Platform.

     Perform the following operations:

     - Call the NetworkInit() and NetworkConnect() functions to establish a TCP connection.
     - Call the MQTTClientInit() function to initialize the MQTT client.
     - Configure the MQTTPacket_connectData structure that contains the MQTT connection parameters.

Sample code:

```
/* network init and establish network to aliyun IoT platform */
NetworkInit(&n);
rc = NetworkConnect(&n, host, port);
printf("NetworkConnect %d\n", rc);
/* init mqtt client */
MQTTClientInit(&c, &n, 1000, buf, sizeof(buf), readbuf, sizeof(readbuf));
/* set the default message handler */
c.defaultMessageHandler = messageArrived;
/* set mqtt connect parameter */
MQTTPacket_connectData data = MQTTPacket_connectData_initializer;
data.willFlag = 0;
data.MQTTVersion = 3;
data.clientID.cstring = clientId;
data.username.cstring = username;
data.password.cstring = password;
data.keepAliveInterval = 60;
data.cleansession = 1;
printf("Connecting to %s %d\n", host, port);
rc = MQTTConnect(&c, &data);
printf("MQTTConnect %d, Connect aliyun IoT Cloud Success!\n", rc);
```

- Publish messages.

  Call the MQTTPublish() function to publish messages in the custom format to the specified custom topic.

  For more information about topics, see What is a topic?.

```
char *pubTopic = "/"EXAMPLE_PRODUCT_KEY"/"EXAMPLE_DEVICE_NAME"/user/update";
int cnt = 0;
unsigned int msgid = 0;
while (!toStop)
{
    MQTTYield(&c, 1000);
    if (++cnt % 5 == 0) {
        MQTTMessage msg = {
            QOS1,
            0,
            0,
            0,
            "Hello world",
            strlen("Hello world"),
        };
        msg.id = ++msgid;
        rc = MQTTPublish(&c, pubTopic, &msg);
        printf("MQTTPublish %d, msgid %d\n", rc, msgid);
    }
}
```

- Subscribe to a topic to receive messages from IoT Platform.

```
void messageArrived(MessageData* md)
{
    MQTTMessage* message = md->message;
    printf("%.*s\t", md->topicName->lenstring.len, md->topicName->lenstring.data);
    printf("%.*s\n", (int)message->payloadlen, (char*)message->payload);
}
char *subTopic = "/"EXAMPLE_PRODUCT_KEY"/"EXAMPLE_DEVICE_NAME"/user/get";
printf("Subscribing to %s\n", subTopic);
rc = MQTTSubscribe(&c, subTopic, 1, messageArrived);
printf("MQTTSubscribe %d\n", rc);
```

For more information about the communication methods of devices, servers, and IoT Platform, see Overview.

3. Copy the *aiot_mqtt_signal.c* file and the file that is edited in Step 2 to *../paho.mqtt.embedded-c/MQTTClient-C/samples/linux* and then compile the project.

Sample code

You can use the sample code to connect with IoT Platform.

1. Download the demo package and decompress it.

   The following files are obtained:

File	Description
aiot_mqtt_sign.c	This file contains the code that is used to obtain the MQTT connection parameters. When you run the *aiot_c_demo.c* file, the aiotMqttSign() function is called to obtain the username, password, and clientId parameters.
aiot_c_demo.c	This file contains the logic code that is used to connect and communicate with IoT Platform.

2. In the *aiot_c_demo.c* file, replace the device information with your device information.
   - Replace the values of the EXAMPLE_PRODUCT_KEY, EXAMPLE_DEVICE_NAME, and EXAMPLE_DEVICE_SECRET parameters with your device certificate information.

     ```
     #define EXAMPLE_PRODUCT_KEY            "The ProductKey of the product"
     #define EXAMPLE_DEVICE_NAME            "The DeviceName of the device"
     #define EXAMPLE_DEVICE_SECRET          "The DeviceSecret of the device"
     ```

   - Modify the endpoint in `char *host = EXAMPLE_PRODUCT_KEY".iot-as-mqtt.cn-shanghai.aliyuncs.com"`.

     Replace cn-shanghai in the code with the ID of the region where your device resides. For more information about region IDs, see Regions and zones.

3. Move the *aiot_mqtt_signal.c* and *aiot_c_demo.c* files to the *../paho.mqtt.embedded-c/MQTTClient-C/samples/linux* directory of the Paho project.
4. Compile the project and run the program.

   You can compile the project by using one of the following methods:

   - Use the **CMake** tool.

a. Add *aiot_c_demo.c* and *aiot_mqtt_signal.c* to the *CMakeLists.txt* file in the */paho.mqtt.embedded-c/MQTTClient-C/samples/linux* directory.

The following code shows how to modify the *CMakeLists.txt* file.

```
add_executable(
  stdoutsubc
  stdoutsub.c
)
add_executable(
  aiot_c_demo
  aiot_c_demo.c
  aiot_mqtt_sign.c
)
target_link_libraries(stdoutsubc paho-embed-mqtt3cc paho-embed-mqtt3c)
target_include_directories(stdoutsubc PRIVATE "../../src" "../../src/linux")
target_compile_definitions(stdoutsubc PRIVATE MQTTCLIENT_PLATFORM_HEADER=MQTTLinu
x.h)
target_link_libraries(aiot_c_demo paho-embed-mqtt3cc paho-embed-mqtt3c)
target_include_directories(aiot_c_demo PRIVATE "../../src" "../../src/linux")
target_compile_definitions(aiot_c_demo PRIVATE MQTTCLIENT_PLATFORM_HEADER=MQTTLin
ux.h)
```

b. Go back to the */paho.mqtt.embedded-c* directory and run the following command to compile the project.

```
mkdir build.paho
cd build.paho
cmake ..
make
```

c. Go to the */paho.mqtt.embedded-c/build.paho* directory and run the following command to run the program:

```
./MQTTClient-C/samples/linux/aiot_c_demo
```

- Use the **build.sh** file.

a. Open the *build.sh* file in the */paho.mqtt.embedded-c/MQTTClient-C/samples/linux* directory.

b. In the *build.sh* file, replace `stdoutsub.c` with `aiot_mqtt_sign.c aiot_c_demo.c`, and `-o stdoutsub` with `-o aiot_c_demo`. Then, save the *build.sh* file.

c. Go to the */paho.mqtt.embedded-c/MQTTClient-C/samples/linux* directory run the `./build.sh` command.

After the compilation, the *aiot_c_demo* executable file is generated.

d. Run the `./aiot_c_demo` command to run the file.

After you run the file, the following local logs are generated:

```
clientid: paho_mqtt&a11xsrW****|timestamp=2524608000000,_v=sdk-c-1.0.0,securemode=3,sig
nmethod=hmacsha256,lan=C|
username: paho_mqtt&a11xsrW****
password: 36E955DC3D9D012EF62C80657A29328B1CFAE6186C611A17DC7939FAB637****
NetworkConnect 0
Connecting to a11xsrW****.iot-as-mqtt.cn-shanghai.aliyuncs.com 443
MQTTConnect 0, Connect aliyun IoT Cloud Success!
Subscribing to /a11xsrW****/paho_mqtt/user/get
MQTTSubscribe 0
MQTTPublish 0, msgid 1
MQTTPublish 0, msgid 2
MQTTPublish 0, msgid 3
MQTTPublish 0, msgid 4
MQTTPublish 0, msgid 5
...
```

Log on to the IoT Platform console.View device status and logs.

- Choose **Devices > Devices**. You can find that the status of the device is **Online**.
- Choose **Maintenance > Device Log**. Then, click the **Cloud run log** or **Device local log** tab to view logs. For more information, see IoT Platform logs and Local device logs.

Error codes

If a device fails to establish an MQTT connection to IoT Platform, you can troubleshoot the issue based on the error code. For more information, see Troubleshooting.

1.1.2. Use the Paho MQTT Java client

This article describes how to use the Paho Message Queuing Telemetry Transport (MQTT) library for Java to connect to IoT Platform and exchange messages with IoT Platform.

Prerequisites

A product and a device are created in the IoT Platform console. The LightSwitch property is defined on the **Define Feature** tab of the Product Details page.

For more information, see Create a product, Create a device, and Add a TSL feature.

Prepare the development environment

In this example, the development environment consists of the following components:

- Operating system: macOS
- Java Development Kit (JDK): JDK 8
- Integrated development environment (IDE): IntelliJ IDEA Community Edition

Download the Paho MQTT library for Java

Add dependencies to the Maven project based on the version of the MQTT protocol.

- MQTT 3.1 and 3.1.1

```
<dependencies>
  <dependency>
      <groupId>org.eclipse.paho</groupId>
      <artifactId>org.eclipse.paho.client.mqttv3</artifactId>
      <version>1.2.0</version>
    </dependency>
</dependencies>
```

- MQTT 5.0

```
<dependency>
   <groupId>org.eclipse.paho</groupId>
   <artifactId>org.eclipse.paho.mqttv5.client</artifactId>
   <version>1.2.5</version>
</dependency>
```

Connect to IoT Platform

1. Click MqttSign.java to obtain the source code provided by Alibaba Cloud to calculate the MQTT connection parameters.

   The *MqttSign.java* file defines the MqttSign class.

   - Prototype

     ```
     class MqttSign
     ```

   - Description

     This class is used to obtain the following MQTT connection parameters: username, password, and clientid.

   - Members

Type	Method
public void	`calculate(String productKey, String deviceName, String deviceSecret)` You can call this method to calculate the username, password, and clientid parameters based on the productKey, deviceName, and deviceSecret of the device.
public String	`getUsername()` You can call this method to obtain the MQTT connection parameter username.
public String	`getPassword()` You can call this method to obtain the MQTT connection parameter password.

Type	Method
public String	`getClientid()` You can call this method to obtain the MQTT connection parameter clientid.

2. Open IntelliJ IDEA and create a project.
3. Import the *MqttSign.java* file into the project.
4. In the project, add a program file that can connect a device to IoT Platform.

   You must write a program to use the MqttSign class in the *MqttSign.java* file to calculate the parameters that are used to establish an MQTT connection to IoT Platform.

   This section provides the development instructions and sample code.

   - Use the MqttSign class to calculate the MQTT connection parameters.

     ```
     String productKey = "a1X2bEn****";
     String deviceName = "example1";
     String deviceSecret = "ga7XA6KdlEeiPXQPpRbAjOZXwG8y****";
     // Calculate the MQTT connection parameters.
     MqttSign sign = new MqttSign();
     sign.calculate(productKey, deviceName, deviceSecret);
     System.out.println("username: " + sign.getUsername());
     System.out.println("password: " + sign.getPassword());
     System.out.println("clientid: " + sign.getClientid());
     ```

   - Create a Paho MQTT client to connect to IoT Platform.

     ```
     // Specify the domain name for connecting to IoT Platform.
     String port = "443";
     String broker = "ssl://" + productKey + ".iot-as-mqtt.cn-shanghai.aliyuncs.com" + ":"
     + port;
     // Create a Paho MQTT client.
     MqttClient sampleClient = new MqttClient(broker, sign.getClientid(), persistence);
     // Set the MQTT connection parameters.
     MqttConnectOptions connOpts = new MqttConnectOptions();
     connOpts.setCleanSession(true);
     connOpts.setKeepAliveInterval(180);
     connOpts.setUserName(sign.getUsername());
     connOpts.setPassword(sign.getPassword().toCharArray());
     sampleClient.connect(connOpts);
     System.out.println("Broker: " + broker + " Connected");
     ```

     > Note

   - Publish messages.

     The following sample code is used to report the LightSwitch property of a TSL model.

```
// Publish messages by using Paho MQTT.
String topic = "/sys/" + productKey + "/" + deviceName + "/thing/event/property/post"
;
String content = "{\"id\":\"1\",\"version\":\"1.0\",\"params\":{\"LightSwitch\":1}}";
MqttMessage message = new MqttMessage(content.getBytes());
message.setQos(0);
sampleClient.publish(topic, message);
```

If you use MQTT 5.0, you can add the following sample code to publish messages that contain custom properties.

```
// MQTT 5.0 new feature: Report custom properties.
MqttProperties properties = new MqttProperties();
List<UserProperty> userPropertys = new ArrayList<>();
userPropertys.add(new UserProperty("key1","value1"));
properties.setUserProperties(userPropertys);
// MQTT 5.0 new feature: Set the request and response mode.
properties.setCorrelationData("requestId12345".getBytes());
properties.setResponseTopic("/" + productKey + "/" + deviceName + "/user/get");
message.setProperties(properties);
// By default, Paho SDK that supports MQTT 5.0 uses topic aliases.
sampleClient.publish(topic, message);
```

For more information about the TSL data format, see Device properties, events, and services.

For more information about how to use custom topics for communication, see What is a topic?

- Subscribe to a topic to obtain messages from IoT Platform.

  In this example, you subscribe to the topic to which IoT Platform returns response messages after the property values are reported.

```
class MqttPostPropertyMessageListener implements IMqttMessageListener {
    @Override
    public void messageArrived(String var1, MqttMessage var2) throws Exception {
        System.out.println("reply topic  : " + var1);
        System.out.println("reply payload: " + var2.toString());
    }
}
...
// Subscribe to a message topic by using Paho MQTT.
String topicReply = "/sys/" + productKey + "/" + deviceName + "/thing/event/property/
post_reply";
sampleClient.subscribe(topicReply, new MqttPostPropertyMessageListener());
```

For more information about the communication methods of devices, servers, and IoT Platform, see Overview.

5. Compile the project.

Demo

The demo code demonstrates how to connect a device to IoT Platform.

1. Download the demo package and decompress it.
2. Open IntelliJ IDEA and import the sample project *aiot-java-demo* in the demo package.

3. In the *src/main/java/com.aliyun.iot* directory, replace the device information with your device information in the *App* or *Mqtt5App* file.

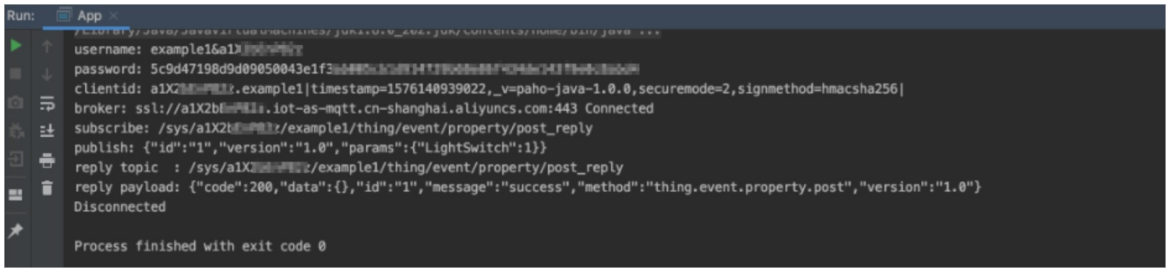
   > **Note** If you use MQTT 3.1 or 3.1.1, replace the device information in the *App* file. If you use MQTT 5.0, replace the device information in the *Mqtt5App* file.

   - Replace the values of the productKey, deviceName, and deviceSecret parameters with your device certificate information.

     ```
     String productKey = "${ProductKey}";
     String deviceName = "${DeviceName}";
     String deviceSecret = "${DeviceSecret}";
     ```

   - Modify the domain name in `String broker = "ssl://" + productKey + ".iot-as-mqtt.cn-shanghai.aliyuncs.com" + ":" + port;`. For more information, see Step 4 in the "Connect to IoT Platform" section.

4. Run the *App* or *Mqtt5App* program.
   The following figure shows the success logs.

   Log on to the IoT Platform console.View device status and logs.

   - Choose **Devices > Devices**. You can find that the status of the device is **Online**.
   - Choose **Maintenance > Device Log** to view logs on the **Cloud run log** and **Device local log** tabs. For more information, see IoT Platform logs and Local device logs.

     If the *Mqtt5App* program is run, you can view the reported custom properties in the log details.

1.1.3. Use the Paho MQTT Go client

This article describes how to use the Paho MQTT library for Go to connect to and communicate with IoT Platform.

Prerequisites

A product and a device are created in the IoT Platform console. The device certificate is obtained. The certificate information includes the ProductKey, DeviceName, and DeviceSecret. For more information, see the following topics:

- View the endpoint of an instance.
- Create a product
- Create a device

Prepare the development environment

Install the Go language package.

- On macOS, run the following command:

```
brew install go
```

- On Ubuntu, run the following command:

```
sudo apt-get install golang-go
```

- On Windows, download the installation package from the Go official website and install the Go language package.

> **Note** If you are in the Chinese mainland, you must use a virtual private network (VPN) to access the Go official website.

Download the Paho MQTT library for Go

For more information about the Paho project and supported programming languages, see Eclipse Paho Downloads.

Run the following commands to download the Paho MQTT library for Go and the related dependencies:

```
go get github.com/eclipse/paho.mqtt.golang
go get github.com/gorilla/websocket
go get golang.org/x/net/proxy
```

Connect the client to IoT Platform

1. Click MqttSign.go and copy the source code that is provided by Alibaba Cloud to obtain the MQTT connection parameters. Then, create the *MqttSign.go* file on premises and paste the code to the file.

   The *MqttSign.go* file defines the function to obtain the MQTT connection parameters. You must call this function to connect your client with IoT Platform.

   - Definition:

     ```
     type AuthInfo struct {
         password, username, mqttClientId string;
     }
     func calculate_sign(clientId, productKey, deviceName, deviceSecret, timeStamp string)
     AuthInfo;
     ```

   - Description:

     This function is used to obtain the following MQTT connection parameters: username, password, and mqttClientId.

   - Input parameters:

Parameter	Type	Description
productKey	String	The ProductKey of the product to which the device belongs. This parameter is used to identify the device in IoT Platform.

Parameter	Type	Description
deviceName	String	The DeviceName of the device. This parameter is used to identify the device in IoT Platform.
deviceSecret	String	The DeviceSecret of the device. This parameter is used to identify the device in IoT Platform.
clientId	String	The ID of the device. The ID can contain up to 64 characters in length. We recommend that you use the MAC address or serial number (SN) of the device as the ID.
timeStamp	String	The timestamp that indicates the current time, in milliseconds.

- Output parameters:

  This function returns an `AuthInfo` structure that contains the parameters that are described in the following table.

Parameter	Type	Description
username	String	The username that is used to establish the MQTT connection.
password	String	The password that is used to establish the MQTT connection.
mqttClientId	String	The ID of the MQTT client.

2. Add a program file that can connect a device to IoT Platform.

   To obtain MQTT connection parameters, you must write a program to call the function in the *Mqtt Sign.go* file.

   This section provides the development instructions and sample code.

   - Specify the device information.

```
// set the device info, include product key, device name, and device secret
    var productKey string = "a1Zd7n5***"
    var deviceName string = "testdevice"
    var deviceSecret string = "UrwclDV33NaFSmk0JaBxNTqgSrJW****"
    // set timestamp, clientid, subscribe topic and publish topic
    var timeStamp string = "1528018257135"
    var clientId string = "192.168.****"
    var subTopic string = "/" + productKey + "/" + deviceName + "/user/get";
    var pubTopic string = "/" + productKey + "/" + deviceName + "/user/update";
```

   - Specify the MQTT connection information.

     Call the calculate_sign() function in *MqttSign.go* file. This function returns the username, password, and mqttClientId parameters based on the clientId, productKey, deviceName, deviceSecret, and timeStamp input parameters. Then, add the result to the opts structure.

```
// set the login broker url
var raw_broker bytes.Buffer
raw_broker.WriteString("tls://")
raw_broker.WriteString(productKey)
raw_broker.WriteString(".iot-as-mqtt.cn-shanghai.aliyuncs.com:1883")
opts := MQTT.NewClientOptions().AddBroker(raw_broker.String());
// calculate the login auth info, and set it into the connection options
auth := calculate_sign(clientId, productKey, deviceName, deviceSecret, timeStamp)
opts.SetClientID(auth.mqttClientId)
opts.SetUsername(auth.username)
opts.SetPassword(auth.password)
opts.SetKeepAlive(60 * 2 * time.Second)
opts.SetDefaultPublishHandler(f)
```

Notice

- If you are using public instances, replace cn-shanghai in `raw_broker.WriteString(".iot-as-mqtt.cn-shanghai.aliyuncs.com:1883")` with the ID of the region in which your device resides. For more information about region IDs, see Regions and zones.
- If you are using Enterprise Edition instances, replace `raw_broker.String()` in `opts := MQTT.NewClientOptions().AddBroker(raw_broker.String());` with `<Endpoint of MQTT>:1833`. Example: `opts := MQTT.NewClientOptions().AddBroker("iot-***.mqtt.iothub.aliyuncs.com:1883");`.

  To obtain the MQTT endpoint, perform the following steps: Log on to the IoT Platform. On the **Overview** page, click the card of the instance that you want to manage. The MQTT endpoint is displayed on the **Instance Details** page. For more information, see View the endpoint of an instance.

For more information about instances, see Overview.

- Call the Connect() function to connect the MQTT client with IoT Platform.

```
// create and start a client using the above ClientOptions
c := MQTT.NewClient(opts)
if token := c.Connect(); token.Wait() && token.Error() != nil {
    panic(token.Error())
}
fmt.Print("Connect aliyun IoT Cloud Success\n");
```

- Call the Publish() function to publish messages. You must specify the topic to which messages are published and the payloads of the messages.

```
// publish 5 messages to pubTopic("/a1Zd7n5****/deng/user/update")
for i := 0; i < 5; i++ {
    fmt.Println("publish msg:", i)
    text := fmt.Sprintf("ABC #%d", i)
    token := c.Publish(pubTopic, 0, false, text)
    fmt.Println("publish msg: ", text)
    token.Wait()
    time.Sleep(2 * time.Second)
}
```

For more information about topics, see What is a topic?.

- Call the Subscribe() function to subscribe to the topic and receive messages from IoT Platform.

```
    // subscribe to subTopic("/a1Zd7n5***/deng/user/get") and request messages to be
delivered
    if token := c.Subscribe(subTopic, 0, nil); token.Wait() && token.Error() != nil {
        fmt.Println(token.Error())
        os.Exit(1)
    }
    fmt.Print("Subscribe topic " + subTopic + " success\n");
```

For more information about the communication methods of devices, servers, and IoT Platform, see Overview.

3. Compile the project.

Sample code

You can use the sample code to connect to IoT Platform.

1. Download and decompress the demo package.

   The following table describes the files that are contained in the *aiot-go-demo* package.

File	Description
MqttSign.go	This file contains the code that is used to obtain the MQTT connection parameters. When you execute the *iot.go* file, the calculate_sign() function is called to obtain the username, password, and mqttClientId parameters.
iot.go	This file contains the logic code that is used to connect to and communicate with IoT Platform.
x509	The root certificate of IoT Platform. This certificate is required to connect devices with IoT Platform.

2. In the *iot.go* file, replace the device information with your device information.

   You can use tools such as Linux vi to modify the *iot.go* file.

   - Replace the values of the productKey, deviceName, and deviceSecret parameters with your device certificate information.
   - Optional. Configure the timeStamp and clientId parameters. You can replace the value of the clientId parameter with the MAC address or SN of your device.

     If you do not configure the two parameters, you can still connect to IoT Platform. However, we recommend that you replace the values of the parameters with actual values.

   - ■ If you are using public instances, replace cn-shanghai in `raw_broker.WriteString(".iot-as-mqtt.cn-shanghai.aliyuncs.com:1883")` with the ID of the region in which your device resides.
     ■ If you are using Enterprise Edition instances, replace `raw_broker.String()` in `opts := MQTT.NewClientOptions().AddBroker(raw_broker.String());` with `<Endpoint of MQTT>:1833`.

   For more information, see Step 2 in the Connect the client to IoT Platform section.

3. Run the following command in the Command Prompt to execute the *iot.go* file:

```
go run iot.go MqttSign.go
```

After you execute the file, the following local logs are generated:

```
clientId192.168.****deviceNametestdeviceproductKeya1Zd7n5****timestamp1528018257135
1b865320fc183cc747041c9faffc9055fc45****
Connect aliyun IoT Cloud Sucess
Subscribe topic /a1Zd7n5****/testdevice/user/get success
publish msg: 0
publish msg:  ABC #0
publish msg: 1
publish msg:  ABC #1
publish msg: 2
publish msg:  ABC #2
publish msg: 3
publish msg:  ABC #3
publish msg: 4
publish msg:  ABC #4
publish msg: 5
publish msg:  ABC #5
```

Log on to the IoT Platform console.View device status and logs.

1.1.4. Use the Paho MQTT C# client

This article describes how to use the Paho Message Queuing Telemetry Transport (MQTT) library for C# to connect to IoT Platform and communicate with IoT Platform.

Prerequisites

A product and a device are created in the IoT Platform console. The LightSwitch property is defined on the **Define Feature** tab of the Product Details page.

For more information, see Create a product, Create a device, and Add a TSL feature.

Context

The open source code of Paho MQTT for C# includes a Visual Studio solution. Each project in the solution can be used to generate the library for the specified .NET platform.

In this example, a console application project is created in the solution. The application uses the Paho MQTT library to connect with IoT Platform.

Prepare the development environment

In this example, the development environment consists of the following components:

- Operating system: Windows 10
- Integrated development environment (IDE): Visual Studio 2019

To install the IDE, perform the following steps:

1. Download Visual Studio Community 2019 and decompress the package.
2. Open *Visual Studio Installer*, click **.NET desktop development**, and then click **Install**.

Download the Paho MQTT C# client

Download the Paho MQTT for C# source code. The code includes a Visual Studio solution file named *M2MMqtt.sln*. You can use this project file to develop your devices. For more information, see Connect the client to IoT Platform.

To view more instructions on the Paho source code, visit the GitHub.

The master branch is used to develop the sample code in this article. The commit ID is `b2e64bc4485721a0bd5ae805d9f4917e8d040e81`.

Connect the client to IoT Platform

1. Click MqttSign.cs to view the source code that is provided by Alibaba Cloud to obtain the MQTT connection parameters.

   The *MqttSign.cs* file defines the `MqttSign` class.

   - Syntax

     ```
     class MqttSign
     ```

   - Description

     This class is used to obtain the following MQTT connection parameters: username, password, and clientid.

   - Members

Type	Method
public bool	`calculate(String productKey, String deviceName, String deviceSecret)` Calculates the username, password, and clientid parameters based on the productKey, deviceName, and deviceSecret parameters.
public String	`getUsername()` Obtains the username parameter.
public String	`getPassword()` Obtains the password parameter.
public String	`getClientid()` Obtains the clientid parameter.

2. Open Visual Studio. Import the Visual Studio solution file *M2Mqtt.sln* and create an application project.
3. Import the *MqttSign.cs* file downloaded in Step 1 into the application project.
4. In the application project, add a program file that can connect a device to IoT Platform.

   You must write a program to use the MqttSign class in the *MqttSign.cs* file to obtain the parameters that are used to establish an MQTT connection with IoT Platform.

   This section provides the development instructions and sample code.

   - Obtain the MQTT connection parameters.

Use the MqttSign class in the *MqttSign.cs* file to obtain the MQTT connection parameters.

```
String productKey = "a1X2bEn****";
String deviceName = "example1";
String deviceSecret = "ga7XA6KdlEeiPXQPpRbAjOZXwG8y****";
// Obtain the MQTT connection parameters.
MqttSign sign = new MqttSign();
sign.calculate(productKey, deviceName, deviceSecret);
Console.WriteLine("username: " + sign.getUsername());
Console.WriteLine("password: " + sign.getPassword());
Console.WriteLine("clientid: " + sign.getClientid());
```

- Create a Paho MQTT client to connect with IoT Platform.

```
// Use the Paho MQTT C# client to connect with IoT Platform.
int port = 443;
String broker = productKey + ".iot-as-mqtt.cn-shanghai.aliyuncs.com";
MqttClient mqttClient = new MqttClient(broker, port, true, MqttSslProtocols.TLSv1_2,
null, null);
mqttClient.Connect(sign.getClientid(), sign.getUsername(), sign.getPassword());
Console.WriteLine("Broker: " + broker + " Connected");
```

> **Note**
>
> Replace cn-shanghai in the code with the ID of the region where your device resides. For more information about region IDs, see Regions and zones.

- Submit data to IoT Platform.

  The following sample code is used to submit the LightSwitch property.

```
// Publish messages by using Paho MQTT.
String topic = "/sys/" + productKey + "/" + deviceName + "/thing/event/property/post"
;
String message = "{\"id\":\"1\",\"version\":\"1.0\",\"params\":{\"LightSwitch\":0}}";
mqttClient.Publish(topic, Encoding.UTF8.GetBytes(message));
```

  For more information about TSL data formats, see Device properties, events, and services.

  For more information about how to use custom topics for communication, see What is a topic?.

- Subscribe to a topic to receive the messages from IoT Platform.

  The following example shows how to subscribe to the topic to which IoT Platform returns a response message after the property is submitted.

```
// Subscribe to a topic by using Paho MQTT.
String topicReply = "/sys/" + productKey + "/" + deviceName + "/thing/event/property/
post_reply";
mqttClient.MqttMsgPublishReceived += MqttPostProperty_MqttMsgPublishReceived;
mqttClient.Subscribe(new string[] { topicReply }, new byte[] { MqttMsgBase.QOS_LEVEL_
AT_MOST_ONCE });
...
private static void MqttPostProperty_MqttMsgPublishReceived(object sender, uPLibrary.
Networking.M2Mqtt.Messages.MqttMsgPublishEventArgs e)
{
    Console.WriteLine("reply topic  :" + e.Topic);
    Console.WriteLine("reply payload:" + e.Message.ToString());
}
```

For more information about the communication methods of devices, servers, and IoT Platform, see Overview.

5. Compile the project.

Sample code

You can use the sample code to connect with IoT Platform.

1. Download the demo package and decompress it to the *aiot-csharp-demo* directory.

   The *aiot-csharp-demo\paho.mqtt.m2mqtt-master\aiot-csharp-demo* directory contains a complete program that is used to connect with IoT Platform and submit properties.

File	Description
MqttSign.cs	This file contains the code that is used to obtain the MQTT connection parameters. When you run the *Program.cs* file, the MqttSign() function is called to obtain the username, password, and clientId parameters.
Program.cs	This file contains the logic code that is used to connect with IoT Platform and submit properties.

2. Open Visual Studio Community 2019, click **Open a project or solution**, and then open the *aiot-csharp-demo\paho.mqtt.m2mqtt-master\M2Mqtt.sln* file.

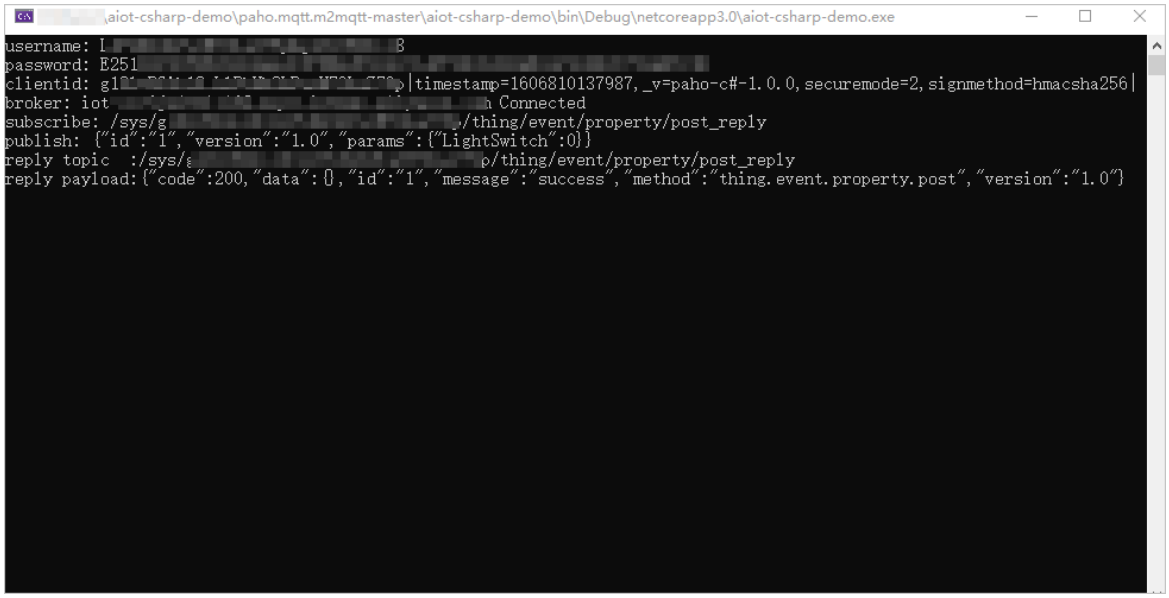
   The *aiot-csharp-demo* project is imported into Visual Studio.

3. In the *Program.cs* file, replace the device information with your device information.
   - Replace the values of the productKey, deviceName, and deviceSecret parameters with your device certificate information.

     ```
     String productKey = "${ProductKey}";
     String deviceName = "${DeviceName}";
     String deviceSecret = "${DeviceSecret}";
     ```

   - Modify the endpoint in `String broker = productKey + ".iot-as-mqtt.cn-shanghai.aliyuncs.com";`. For more information, see Step 4 in the "Connect the client to IoT Platform" section.

4. Set the *aiot-csharp-demo* file as the startup item, and then run it to connect the client with IoT Platform.



After the connection is established, you can view the information about the connection, data submission, and message subscription in local logs.

```
...
broker: a1X2bEn****.iot-as-mqtt.cn-shanghai.aliyuncs.com Connected
...
publish: {"id":"1","version":"1.0","params":{"LightSwitch":0}}
...
subscribe: /sys/a1X2bEn****/example1/thing/event/property/post_reply
...
```

Log on to the IoT Platform console.View device status and logs.

1.1.5. Use the Paho MQTT Android client

This article describes how to use Paho Android Service to connect to IoT Platform and exchange messages with IoT Platform.

Prerequisites

Context

Paho Android Service is a Message Queuing Telemetry Transport (MQTT) client that is developed based on the Paho MQTT library for Java.

Prepare the development environment

In this example, Android Studio of version 3.5.1 and Gradle of version 3.5.1 are used.

Visit the Android Studio official website to download Android Studio. For more information about Android development, see the Android Studio official documentation.

Install Paho Android Client

1. Create an Android project.
2. In the Gradle file, add the Paho Android Client dependencies. In this example, Paho Android Client

1.1.1 is used. You need to add the following dependencies.

- In the *build.gradle* file, add the address of the Paho repository. In this example, the release repository is used.

```
repositories {
    maven {
        url "https://repo.eclipse.org/content/repositories/paho-releases/"
    }
}
```

- In the *build.gradle* file, add the Paho Android Service dependencies. In this example, Paho Android Service 1.1.1 is used. It is based on paho.client.mqttv3-1.1.0.

```
dependencies {
    implementation 'org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.1.0'
    implementation 'org.eclipse.paho:org.eclipse.paho.android.service:1.1.1'
}
```

3. To bind an app to Paho Android Service, add the following information to the *AndroidManifest.xml* file.
   - Declare the following service:

```
<! -- Mqtt Service -->
<service android:name="org.eclipse.paho.android.service.MqttService">
</service>
```

   - Add the permissions required by Paho MQTT Service.

```
<uses-permission android:name="android.permission.WAKE_LOCK" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
```

Connect to IoT Platform

1. Click AiotMqttOption.java to obtain the source code provided by Alibaba Cloud to calculate the MQTT connection parameters.

   The *AiotMqttOption.java* file defines the AiotMqttOption class.

   - Prototype

```
class AiotMqttOption
```

   - Description

     This class is used to obtain the following MQTT connection parameters: username, password, and clientid.

   - Members

Type	Method

Type	Method
public AiotMqttOption	`getMqttOption(String productKey, String deviceName, String deviceSecret)` You can call this method to calculate the username, password, and clientid parameters based on the productKey, deviceName, and deviceSecret of the device.
public String	`getUsername()` You can call this method to obtain the MQTT connection parameter username.
public String	`getPassword()` You can call this method to obtain the MQTT connection parameter password.
public String	`getClientid()` You can call this method to obtain the MQTT connection parameter clientid.

2. Import the *AiotMqttOption.java* file into the Android project.
3. In the Android project, add a program file that can connect a device to IoT Platform.

   You must write a program to use the AiotMqttOption class in the *AiotMqttOption.java* file to calculate the parameters that are used to establish an MQTT connection to IoT Platform.

   This section provides the development instructions and sample code.

   - Calculate the MQTT connection parameters clientId, username, and password. Configure the username and password parameters in the MqttConnectOptions object.

```
final private String PRODUCTKEY = "a11xsrW****";
final private String DEVICENAME = "paho_android";
final private String DEVICESECRET = "tLMT9QWD36U2SArglGqcHCDK9rK9****";
/* Obtain the MQTT connection parameters clientId, username, and password. */
AiotMqttOption aiotMqttOption = new AiotMqttOption().getMqttOption(PRODUCTKEY, DEVICE
NAME, DEVICESECRET);
if (aiotMqttOption == null) {
    Log.e(TAG, "device info error");
} else {
    clientId = aiotMqttOption.getClientId();
    userName = aiotMqttOption.getUsername();
    passWord = aiotMqttOption.getPassword();
}
/* Create an MqttConnectOptions object and configure the username and password. */
MqttConnectOptions mqttConnectOptions = new MqttConnectOptions();
mqttConnectOptions.setUserName(userName);
mqttConnectOptions.setPassword(passWord.toCharArray());
```

   - Connect to IoT Platform.

Create an MqttAndroidClient object and configure the callback. Use the mqttConnectOptions object to call the connect() method to establish the connection.

```
/* Create an MqttAndroidClient object and configure the callback. */
mqttAndroidClient = new MqttAndroidClient(getApplicationContext(), host, clientId);
mqttAndroidClient.setCallback(new MqttCallback() {
    @Override
    public void connectionLost(Throwable cause) {
        Log.i(TAG, "connection lost");
    }
    @Override
    public void messageArrived(String topic, MqttMessage message) throws Exception {
        Log.i(TAG, "topic: " + topic + ", msg: " + new String(message.getPayload()));
    }
    @Override
    public void deliveryComplete(IMqttDeliveryToken token) {
        Log.i(TAG, "msg delivered");
    }
});
/* Establish a connection to IoT Platform by using MQTT. */
try {
    mqttAndroidClient.connect(mqttConnectOptions, null, new IMqttActionListener() {
        @Override
        public void onSuccess(IMqttToken asyncActionToken) {
            Log.i(TAG, "connect succeed");
            subscribeTopic(SUB_TOPIC);
        }
        @Override
        public void onFailure(IMqttToken asyncActionToken, Throwable exception) {
            Log.i(TAG, "connect failed");
        }
    });
} catch (MqttException e) {
    e.printStackTrace();
}
```

- Publish messages. Write the publishMessage() method to publish messages with the specified payload to the `/${prodcutKey}/${deviceName}/user/update` topic.

```
public void publishMessage(String payload) {
    try {
        if (mqttAndroidClient.isConnected() == false) {
            mqttAndroidClient.connect();
        }
        MqttMessage message = new MqttMessage();
        message.setPayload(payload.getBytes());
        message.setQos(0);
        mqttAndroidClient.publish(PUB_TOPIC, message,null, new IMqttActionListener()
{
            @Override
            public void onSuccess(IMqttToken asyncActionToken) {
                Log.i(TAG, "publish succeed!") ;
            }
            @Override
            public void onFailure(IMqttToken asyncActionToken, Throwable exception) {
                Log.i(TAG, "publish failed!") ;
            }
        });
    } catch (MqttException e) {
        Log.e(TAG, e.toString());
        e.printStackTrace();
    }
}
```

For more information about communication topics, see What is a topic?

- Write the subscribeTopic() method to subscribe to a specified topic and obtain messages from IoT Platform.

```
public void subscribeTopic(String topic) {
    try {
        mqttAndroidClient.subscribe(topic, 0, null, new IMqttActionListener() {
            @Override
            public void onSuccess(IMqttToken asyncActionToken) {
                Log.i(TAG, "subscribed succeed");
            }
            @Override
            public void onFailure(IMqttToken asyncActionToken, Throwable exception) {
                Log.i(TAG, "subscribed failed");
            }
        });
    } catch (MqttException e) {
        e.printStackTrace();
    }
}
```

For more information about the communication methods of devices, servers, and IoT Platform, see Overview.

4. Compile the project.

Demo

The demo code demonstrates how to connect a device to IoT Platform.

1. Download the demo package and decompress it.
2. Import *aiot-android-demo* into Android Studio.
3. In the *app/src/main/java/com.linkkit.aiot_android_demo* directory, replace the device information with your device information in the *MainActivity* file.
   - Replace the values of the productKey, deviceName, and deviceSecret parameters with your device certificate information.
   - Replace cn-shanghai in `final String host = "tcp://" + PRODUCTKEY + ".iot-as-mqtt.cn-shanghai.aliyuncs.com:443";` with the ID of the region where your device resides. For more information about region IDs, see Regions and zones.
4. Build and run the demo.
   After the demo is run, you can view the local logs in Logcat.

```
2019-12-04 19:44:01.824 5952-5987/com.linkkit.aiot_android_demo W/OpenGLRenderer: Faile
d to choose config with EGL_SWAP_BEHAVIOR_PRESERVED, retrying without...
2019-12-04 19:44:01.829 5952-5987/com.linkkit.aiot_android_demo D/EGL_emulation: eglCre
ateContext: 0xec073240: maj 3 min 0 rcv 3
2019-12-04 19:44:01.830 5952-5987/com.linkkit.aiot_android_demo D/EGL_emulation: eglMak
eCurrent: 0xec073240: ver 3 0 (tinfo 0xec09b470)
2019-12-04 19:44:01.852 5952-5987/com.linkkit.aiot_android_demo W/Gralloc3: mapper 3.x
is not supported
2019-12-04 19:44:01.854 5952-5987/com.linkkit.aiot_android_demo D/HostConnection: creat
eUnique: call
...
...
2019-12-04 19:44:01.860 5952-5987/com.linkkit.aiot_android_demo D/eglCodecCommon: alloc
ate: Ask for block of size 0x1000
2019-12-04 19:44:01.861 5952-5987/com.linkkit.aiot_android_demo D/eglCodecCommon: alloc
ate: ioctl allocate returned offset 0x3ff706000 size 0x2000
2019-12-04 19:44:01.897 5952-5987/com.linkkit.aiot_android_demo D/EGL_emulation: eglMak
eCurrent: 0xec073240: ver 3 0 (tinfo 0xec09b470)
2019-12-04 19:44:02.245 5952-6023/com.linkkit.aiot_android_demo D/AlarmPingSender: Regi
ster alarmreceiver to MqttServiceMqttService.pingSender.a11xsrW****.paho_android|timest
amp=1575459841629,_v=sdk-android-1.0.0,securemode=2,signmethod=hmacsha256|
2019-12-04 19:44:02.256 5952-6023/com.linkkit.aiot_android_demo D/AlarmPingSender: Sche
dule next alarm at 1575459902256
2019-12-04 19:44:02.256 5952-6023/com.linkkit.aiot_android_demo D/AlarmPingSender: Alar
m scheule using setExactAndAllowWhileIdle, next: 60000
2019-12-04 19:44:02.272 5952-5952/com.linkkit.aiot_android_demo I/AiotMqtt: connect suc
ceed
2019-12-04 19:44:02.301 5952-5952/com.linkkit.aiot_android_demo I/AiotMqtt: subscribed
succeed
```

   Log on to the IoT Platform console.View device status and logs.
   - Choose **Devices > Devices**. You can find that the status of the device is **Online**.
   - Choose **Maintenance > Device Log** to view logs on the **Cloud run log** and **Device local log** tabs. For more information, see IoT Platform logs and Local device logs.

     If the *Mqtt5App* program is run, you can view the reported custom properties in the log details.

1.1.6. Troubleshooting

If a device fails to establish a Message Queuing Telemetry Transport (MQTT) connection to IoT Platform, you can troubleshoot the issue based on the error code.

IoT Platform uses the standard MQTT protocol. For more information about the MQTT protocol, see the MQTT 3.1 or 3.1.1 documentation and MQTT 5.0 documentation.

The following tables describe the codes that may be returned by IoT Platform.

- MQTT 3.1 and 3.1.1

Return code	Return message	Description
0	0x00 Connection Accepted	The message returned when the connection is established.
1	0x01 Connection Refused, unacceptable protocol version	The error message returned because the server does not support the MQTT protocol version that is used by the device.
2	0x02 Connection Refused, identifier rejected	The error message returned because the server does not support the UTF-8 encoded client ID.
3	0x03 Connection Refused, Server unavailable	The error message returned because the MQTT service is unavailable even though the network connection has been established.
4	0x04 Connection Refused, bad user name or password	The error message returned because the format of the username or password parameter is invalid.
5	0x05 Connection Refused, not authorized	The error message returned because the device is not authorized.

- MQTT 5.0

Return code	Return message	Description
0	0x00 Success	The message returned when the connection is established.
128	0x80 Unspecified error	The error message returned because an unspecified error occurred.
129	0x81 Malformed Packet	The error message returned because a malformed packet was received.
130	0x82 Protocol Error	The error message returned because a protocol error occurred.
132	0x84 Unsupported Protocol Version	The error message returned because the protocol version is not supported.

Return code	Return message	Description
136	0x88 Server unavailable	The error message returned because the server is unavailable.
137	0x89 Server busy	The error message returned because the server is busy.
138	0x8A Banned	The error message returned because the access is prohibited.
140	0x8C Bad authentication method	The error message returned because the authentication method is invalid.
141	0x8D Keep Alive timeout	The error message returned because a keep-alive timeout occurred.
144	0x90 Topic Name invalid	The error message returned because the topic name is invalid.
147	0x93 Receive Maximum exceeded	The error message returned because the number of received messages exceeds the limit.
148	0x94 Topic Alias invalid	The error message returned because the topic alias is invalid.
149	0x95 Packet too large	The error message returned because the packet length exceeds the limit.
150	0x96 Message rate too high	The error message returned because the message transmission rate is too high.
151	0x97 Quota exceeded	The error message returned because the quota is exceeded.
152	0x98 Administrative action	The error message returned because of an administrative action.
153	0x99 Payload format invalid	The error message returned because the payload format is invalid.
154	0x9A Retain not supported	The error message returned because messages cannot be retained.
155	0x9B QoS not supported	The error message returned because the QoS is not supported.
156	0x9C Use another server	The error message returned because another server needs to be used.
157	0x9D Server moved	The error message returned because the server is removed.

Return code	Return message	Description
158	0x9E Shared Subscription not supported	The error message returned because shared subscription is not supported.
159	0x9F Connection rate exceeded	The error message returned because the connection rate exceeds the limit.

1.2. Access IoT Platform by using CoAP

This topic describes how to connect a device to IoT Platform by using Constrained Application Protocol (CoAP).

CoAP is suitable for devices such as NB-IoT devices that have constrained resources and run at low power. For more information about how to connect a device to IoT Platform by using CoAP, see Connect devices to IoT Platform over CoAP.

> **Note** The feature is available only in the China (Shanghai) region.

When you connect a device to IoT Platform by using CoAP, you must specify the required parameters, generate a signature for the device, and perform other operations. Eclipse Californium-based sample code is used as an example to describe the configuration process. To ensure data security, symmetric encryption is applied.

pom.xml

Add the following dependencies to the *pom.xml* file to import the Californium open-source framework, Apache Commons toolkit, and Alibaba fastjson package.

```
<dependency>
  <groupId>org.eclipse.californium</groupId>
  <artifactId>californium-core</artifactId>
  <version>2.0.0-M17</version>
</dependency>
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>3.5</version>
</dependency>
<dependency>
  <groupId>commons-codec</groupId>
  <artifactId>commons-codec</artifactId>
  <version>1.13</version>
</dependency>
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>fastjson</artifactId>
  <version>1.2.61</version>
</dependency>
```

Sample code

```
/*
 * Copyright © 2019 Alibaba. All rights reserved.
 */
package com.aliyun.iot.demo;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Set;
import java.util.SortedMap;
import java.util.TreeMap;
import javax.crypto.Cipher;
import javax.crypto.Mac;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import org.apache.commons.codec.DecoderException;
import org.apache.commons.codec.binary.Hex;
import org.apache.commons.lang3.RandomUtils;
import org.eclipse.californium.core.CoapClient;
import org.eclipse.californium.core.CoapResponse;
import org.eclipse.californium.core.Utils;
import org.eclipse.californium.core.coap.CoAP;
import org.eclipse.californium.core.coap.CoAP.Code;
import org.eclipse.californium.core.coap.CoAP.Type;
import org.eclipse.californium.core.coap.MediaTypeRegistry;
import org.eclipse.californium.core.coap.Option;
import org.eclipse.californium.core.coap.OptionNumberRegistry;
import org.eclipse.californium.core.coap.OptionSet;
import org.eclipse.californium.core.coap.Request;
import org.eclipse.californium.elements.exception.ConnectorException;
import com.alibaba.fastjson.JSONObject;
/**
 * The following Eclipse Californium-based code shows how to connect a device to IoT Platfo
rm by using CoAP.
 * For more information about how to develop custom code and specify the required parameter
s, see the Use the symmetric encryption method Section in Establish connections over CoAP.
 */
public class IotCoapClientWithAes {
    // ====================The start of the section where you can specify the required param
eters============================
    // The ID of the region. Only the China (Shanghai) region is available.
    private static String regionId = "cn-shanghai";
    // The key of the product.
    private static String productKey = "The productKey of your device";
    // The name of the device.
    private static String deviceName = "The deviceName of your device";
    // The key of the device.
    private static String deviceSecret = "The deviceSecret of your device";
    // ====================The end of the section where you can specify the required paramet
ers============================
    // Set the HMAC_ALGORITHM parameter to hmacsha1 or hmacmd5. The value that you set must
be the same as that of the signmethod parameter.
```

```
    private static final String HMAC_ALGORITHM = "hmacsha1";
    // The endpoint that is used to authenticate CoAP clients. The port number of the endpo
int is 5682 if you use symmetric encryption.
    private static String serverURI = "coap://" + productKey + ".coap." + regionId + ".link
.aliyuncs.com:5682";
    // The topic to which messages are sent. In the IoT Platform console, you can create a
custom topic and grant the publish permission to devices.
    private static String updateTopic = "/" + productKey + "/" + deviceName + "/user/update
";
    // token option
    private static final int COAP2_OPTION_TOKEN = 2088;
    // seq option
    private static final int COAP2_OPTION_SEQ = 2089;
    // Specify the SHA-256 encryption algorithm.
    private static final String SHA_256 = "SHA-256";
    private static final int DIGITAL_16 = 16;
    private static final int DIGITAL_48 = 48;
    // Create a CoAP client.
    private CoapClient coapClient = new CoapClient();
    // The validity period of a token is seven days. After a token expires, you must obtain
a new token.
    private String token = null;
    private String random = null;
    @SuppressWarnings("unused")
    private long seqOffset = 0;
    /**
     * Initialize a CoAP client
     *
     * @param productKey The key of the product
     * @param deviceName The name of the device
     * @param deviceSecret The key of the device
     */
    public void connect(String productKey, String deviceName, String deviceSecret) {
        try {
            // The endpoint for authentication.
            String uri = serverURI + "/auth";
            // Only the POST method is available.
            Request request = new Request(Code.POST, Type.CON);
            // Specify options
            OptionSet optionSet = new OptionSet();
            optionSet.addOption(new Option(OptionNumberRegistry.CONTENT_FORMAT, MediaTypeRe
gistry.APPLICATION_JSON));
            optionSet.addOption(new Option(OptionNumberRegistry.ACCEPT, MediaTypeRegistry.A
PPLICATION_JSON));
            request.setOptions(optionSet);
            // Specify the endpoint that you can use to authenticate the device
            request.setURI(uri);
            // Specify the required parmeters for an authentication request
            request.setPayload(authBody(productKey, deviceName, deviceSecret));
            // Send the authentication request
            CoapResponse response = coapClient.advanced(request);
            System.out.println(Utils.prettyPrint(response));
            System.out.println();
            // Parse the response
```

```
            JSONObject json = JSONObject.parseObject(response.getResponseText());
            token = json.getString("token");
            random = json.getString("random");
            seqOffset = json.getLongValue("seqOffset");
        } catch (ClientException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    /**
     * Send messages
     *
     @param topic The topic to which messages are sent.
     * @param payload The contents of messages.
     */
    public void publish(String topic, byte[] payload) {
        try {
            // The endponit of the topic. The syntax of a topic is /topic/${topic}.
            String uri = serverURI + "/topic" + topic;
            // Use  seq=RandomUtils.nextInt() to encrypt the seq option by using the AES en
cryption algorithm.
            String shaKey = encod(deviceSecret + "," + random);
            byte[] keys = Hex.decodeHex(shaKey.substring(DIGITAL_16, DIGITAL_48));
            byte[] seqBytes = encrypt(String.valueOf(RandomUtils.nextInt()).getBytes(Standa
rdCharsets.UTF_8), keys);
            // Only the POST method is available
            Request request = new Request(CoAP.Code.POST, CoAP.Type.CON);
            // Specify options
            OptionSet optionSet = new OptionSet();
            optionSet.addOption(new Option(OptionNumberRegistry.CONTENT_FORMAT, MediaTypeRe
gistry.APPLICATION_JSON));
            optionSet.addOption(new Option(OptionNumberRegistry.ACCEPT, MediaTypeRegistry.A
PPLICATION_JSON));
            optionSet.addOption(new Option(COAP2_OPTION_TOKEN, token));
            optionSet.addOption(new Option(COAP2_OPTION_SEQ, seqBytes));
            request.setOptions(optionSet);
            // Specify the endpoint of the topic.
            request.setURI(uri);
            // Specify the payload parmameter.
            request.setPayload(encrypt(payload, keys));
            // Send messages.
            CoapResponse response = coapClient.advanced(request);
            System.out.println(Utils.prettyPrint(response));
            // Parse the result.
            String result = null;
            if (response.getPayload() != null) {
                result = new String(decrypt(response.getPayload(), keys));
            }
            System.out.println("Received: " + result);
            System.out.println();
        } catch (ConnectorException e) {
            e.printStackTrace();
        } catch (IOException e) {
```

```
            e.printStackTrace();
        } catch (DecoderException e) {
            e.printStackTrace();
        }
    }
    /**
     * Generate the required parameters for authentication
     *
     * @param productKey The name of the product
     * @param deviceName The name of the device
     * @param deviceSecret The key of the device
     * @return An authentication request
     */
    private String authBody(String productKey, String deviceName, String deviceSecret) {
        // Create an authentication request
        JSONObject body = new JSONObject();
        body.put("productKey", productKey);
        body.put("deviceName", deviceName);
        body.put("clientId", productKey + "." + deviceName);
        //signMap.put("timestamp", String.valueOf(System.currentTimeMillis()));
        body.put("signmethod", HMAC_ALGORITHM);
        body.put("seq", DIGITAL_16);
        body.put("sign", sign(body, deviceSecret));
        System.out.println("----- auth body -----");
        System.out.println(body.toJSONString());
        return body.toJSONString();
    }
    /**
     * Generate a signature for a device
     *
     * @param params The required parameters that you can use to generate a signature.
     * @param deviceSecret The key of the device.
     * @return The signature in the hexadecimal format.
     */
    private String sign(JSONObject params, String deviceSecret) {
        // Sort request parameters in the alphabetical order
        Set<String> keys = getSortedKeys(params);
        //  Remove the sign, signmethod, version, and resources parameters.
        keys.remove("sign");
        keys.remove("signmethod");
        keys.remove("version");
        keys.remove("resources");
        // Obtain the plaintext of the signature.
        StringBuffer content = new StringBuffer();
        for (String key : keys) {
            content.append(key);
            content.append(params.getString(key));
        }
        // Generate a signature.
        String sign = encrypt(content.toString(), deviceSecret);
        System.out.println("sign content=" + content);
        System.out.println("sign result=" + sign);
        return sign;
    }
    /**
```

```
     * Convert a JSON object to a set of key-value pairs.
     *
     * @param json The JSON object to be converted.
     * @return A set of key-value pairs that are converted from a JSON object.
     */
    private Set<String> getSortedKeys(JSONObject json) {
        SortedMap<String, String> map = new TreeMap<String, String>();
        for (String key : json.keySet()) {
            String vlaue = json.getString(key);
            map.put(key, vlaue);
        }
        return map.keySet();
    }
    /**
     * Specify an encryption algorithm in the HMAC_ALGORITHM parameter.
     *
     * @param content Plaintext.
     * @param secret An encryption key.
     * @return Ciphertext.
     */
    private String encrypt(String content, String secret) {
        try {
            byte[] text = content.getBytes(StandardCharsets.UTF_8);
            byte[] key = secret.getBytes(StandardCharsets.UTF_8);
            SecretKeySpec secretKey = new SecretKeySpec(key, HMAC_ALGORITHM);
            Mac mac = Mac.getInstance(secretKey.getAlgorithm());
            mac.init(secretKey);
            return Hex.encodeHexString(mac.doFinal(text));
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
    /**
     * SHA-256
     *
     * @param str A message to be encrypted.
     */
    private String encod(String str) {
        MessageDigest messageDigest;
        String encdeStr = "";
        try {
            messageDigest = MessageDigest.getInstance(SHA_256);
            byte[] hash = messageDigest.digest(str.getBytes(StandardCharsets.UTF_8));
            encdeStr = Hex.encodeHexString(hash);
        } catch (NoSuchAlgorithmException e) {
            System.out.println(String.format("Exception@encod: str=%s;", str));
            e.printStackTrace();
            return null;
        }
        return encdeStr;
    }
    // Encrypt and decrypt data based on the AES algorithm.
    private static final String IV = "543yhjy97ae7fyfg";
```

```
    private static final String TRANSFORM = "AES/CBC/PKCS5Padding";
    private static final String ALGORITHM = "AES";
    /**
     * key length = 16 bits
     */
    private byte[] encrypt(byte[] content, byte[] key) {
        return encrypt(content, key, IV);
    }
    /**
     * key length = 16 bits
     */
    private byte[] decrypt(byte[] content, byte[] key) {
        return decrypt(content, key, IV);
    }
    /**
     * aes 128 cbc key length = 16 bits
     */
    private byte[] encrypt(byte[] content, byte[] key, String ivContent) {
        try {
            SecretKeySpec keySpec = new SecretKeySpec(key, ALGORITHM);
            Cipher cipher = Cipher.getInstance(TRANSFORM);
            IvParameterSpec iv = new IvParameterSpec(ivContent.getBytes(StandardCharsets.UT
F_8));
            cipher.init(Cipher.ENCRYPT_MODE, keySpec, iv);
            return cipher.doFinal(content);
        } catch (Exception ex) {
            System.out.println(
                    String.format("AES encrypt error, %s, %s, %s", content, Hex.encodeHex(k
ey), ex.getMessage()));
            return null;
        }
    }
    /**
     * aes 128 cbc key length = 16 bits
     */
    private byte[] decrypt(byte[] content, byte[] key, String ivContent) {
        try {
            SecretKeySpec keySpec = new SecretKeySpec(key, ALGORITHM);
            Cipher cipher = Cipher.getInstance(TRANSFORM);
            IvParameterSpec iv = new IvParameterSpec(ivContent.getBytes(StandardCharsets.UT
F_8));
            cipher.init(Cipher.DECRYPT_MODE, keySpec, iv);
            return cipher.doFinal(content);
        } catch (Exception ex) {
            System.out.println(String.format("AES decrypt error, %s, %s, %s", Hex.encodeHex
(content),
                    Hex.encodeHex(key), ex.getMessage()));
            return null;
        }
    }
    public static void main(String[] args) throws InterruptedException {
        IotCoapClientWithAes client = new IotCoapClientWithAes();
        client.connect(productKey, deviceName, deviceSecret);
        client.publish(updateTopic, "hello coap".getBytes(StandardCharsets.UTF_8));
```

```
        client.publish(updateTopic, new byte[] { 0x01, 0x02, 0x03, 0x05 });
    }
}
```

1.3. Access IoT Platform by using HTTP

This topic describes how to connect a device to IoT Platform by using HTTP.

You can connect a device to IoT Platform by using HTTP. For more information, see Establish connections over HTTPS.

This topic uses a sample code developed based on the device SDK for Java to connect a device with IoT Platform. Before you can establish the connectioin, you must specify the required parameters, such as request parameters and certificate information of the device.

> **Note** This feature is available only in the China (Shanghai) region.

Configure the pom.xml file

Add the following dependency to the *pom.xml* file to import the Alibaba fastjson package.

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>fastjson</artifactId>
  <version>1.2.61</version>
</dependency>
```

Sample code

The following code shows how to connect a device to IoT Platform and enable data communication.

```
/*
 * Copyright © 2019 Alibaba. All rights reserved.
 */
package com.aliyun.iot.demo;
import java.io.BufferedOutputStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.URL;
import java.nio.charset.StandardCharsets;
import java.util.Set;
import java.util.SortedMap;
import java.util.TreeMap;
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import javax.net.ssl.HttpsURLConnection;
import com.alibaba.fastjson.JSONObject;
/**
 * Connect a device to IoT Platform by using HTTP.
```

```
 * For more information about the HTTP protocol, see HTTP standard.
 * For more information about the data format, see Establish connections over HTTP.
 */
public class IotHttpClient {
    // The ID of the region. Only the China (Shanghai) region is available.
    private static String regionId = "cn-shanghai";
    // Specify an encryption algorithm. Set the HMAC_ALGORITHM parameter to hmacsha1 or hma
cmd5. The value that you set must be the same as that of the signmethod parameter.
    private static final String HMAC_ALGORITHM = "hmacsha1";
    // The validity period of a token is seven days. After a token expires, you must obtain
a new token.
    private String token = null;
    /**
     * Initialize an HTTP client.
     *
     * @param productKey The key of the product.
     * @param deviceName The name of the device.
     * @param deviceSecret The key of the device.
     */
    public void conenct(String productKey, String deviceName, String deviceSecret) {
        try {
            // The endpoint for authentication.
            URL url = new URL("https://iot-as-http." + regionId + ".aliyuncs.com/auth");
            HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
            conn.setRequestMethod("POST");
            conn.setRequestProperty("Content-type", "application/json");
            conn.setDoOutput(true);
            conn.setDoInput(true);
            // Obtain the output stream of the URLConnection object.
            PrintWriter out = new PrintWriter(conn.getOutputStream());
            // Send request parameters.
            out.print(authBody(productKey, deviceName, deviceSecret));
            // Print out the contents of the cache.
            out.flush();
            // Obtain the input stream of the URLConnection object.
            BufferedReader in = new BufferedReader(new InputStreamReader(conn.getInputStrea
m()));
            // Obtain responses from the endpoint.
            String result = "";
            String line = "";
            while ((line = in.readLine()) != null) {
                result += line;
            }
            System.out.println("----- auth result -----");
            System.out.println(result);
            // Close input and output streams.
            in.close();
            out.close();
            conn.disconnect();
            // Obtain a token.
            JSONObject json = JSONObject.parseObject(result);
            if (json.getIntValue("code") == 0) {
                token = json.getJSONObject("info").getString("token");
            }
        } catch (Exception e) {
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    /**
     * Send messages.
     *
     @param topic The topic to which messages are sent.
     * @param payload The contents of messages.
     */
    public void publish(String topic, byte[] payload) {
        try {
            // Specify the endpoint of a topic to which messages are sent.
            URL url = new URL("https://iot-as-http." + regionId + ".aliyuncs.com/topic" + t
opic);
            HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
            conn.setRequestMethod("POST");
            conn.setRequestProperty("Content-type", "application/octet-stream");
            conn.setRequestProperty("password", token);
            conn.setDoOutput(true);
            conn.setDoInput(true);
            // Obtain the output stream of the URLConnection object.
            BufferedOutputStream out = new BufferedOutputStream(conn.getOutputStream());
            out.write(payload);
            out.flush();
            // Obtain the input stream of the URLConnection object.
            BufferedReader in = new BufferedReader(new InputStreamReader(conn.getInputStrea
m()));
            // Obtain responses from the endpoint.
            String result = "";
            String line = "";
            while ((line = in.readLine()) != null) {
                result += line;
            }
            System.out.println("----- publish result -----");
            System.out.println(result);
            // Close input and output streams.
            in.close();
            out.close();
            conn.disconnect();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    /**
     * Generate the required parameters for authentication.
     *
     * @param params The required parameters for authentication.
     * @return A message that includes an authentication request.
     */
    private String authBody(String productKey, String deviceName, String deviceSecret) {
        // Create a authentication request.
        JSONObject body = new JSONObject();
        body.put("productKey", productKey);
        body.put("deviceName", deviceName);
```

```
        body.put("clientId", productKey + "." + deviceName);
        body.put("timestamp", String.valueOf(System.currentTimeMillis()));
        body.put("signmethod", HMAC_ALGORITHM);
        body.put("version", "default");
        body.put("sign", sign(body, deviceSecret));
        System.out.println("----- auth body -----");
        System.out.println(body.toJSONString());
        return body.toJSONString();
    }
    /**
     * Generate a signature for the device
     *
     * @param params The required parameters that you can use to generate a signature
     * @param deviceSecret The key of a device
     * @return The signature in the hexadecimal format.
     */
    private String sign(JSONObject params, String deviceSecret) {
        // Sort request parameters in alphbetical order.
        Set<String> keys = getSortedKeys(params);
        // Remove the sign, signmethod, and version parameters.
        keys.remove("sign");
        keys.remove("signmethod");
        keys.remove("version");
        // Obtain the plaintext of the signature
        StringBuffer content = new StringBuffer();
        for (String key : keys) {
            content.append(key);
            content.append(params.getString(key));
        }
        // Generate a signature
        String sign = encrypt(content.toString(), deviceSecret);
        System.out.println("sign content=" + content);
        System.out.println("sign result=" + sign);
        return sign;
    }
    /**
     * Convert a JSON object to a set of key-value pairs.
     *
     * @param json The JSON object to be converted.
     * @return A set of key-value pairs that are converted from the JSON object.
     */
    private Set<String> getSortedKeys(JSONObject json) {
        SortedMap<String, String> map = new TreeMap<String, String>();
        for (String key : json.keySet()) {
            String vlaue = json.getString(key);
            map.put(key, vlaue);
        }
        return map.keySet();
    }
    /**
     * Specify an encryption method in the HMAC_ALGORITHM parameter.
     *
     * @param content Plaintext.
     * @param secret An encryption key.
```

```
     * @return Ciphertext.
     */
    private String encrypt(String content, String secret) {
        try {
            byte[] text = content.getBytes(StandardCharsets.UTF_8);
            byte[] key = secret.getBytes(StandardCharsets.UTF_8);
            SecretKeySpec secretKey = new SecretKeySpec(key, HMAC_ALGORITHM);
            Mac mac = Mac.getInstance(secretKey.getAlgorithm());
            mac.init(secretKey);
            return byte2hex(mac.doFinal(text));
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
    /**
     * Convert a binary array to a hexadecimal string.
     *
     * @param b The binary array.
     * @return A hexadecimal string.
     */
    private String byte2hex(byte[] b) {
        StringBuffer sb = new StringBuffer();
        for (int n = 0; b != null && n < b.length; n++) {
            String stmp = Integer.toHexString(b[n] & 0XFF);
            if (ss.length == 1) {
                sb.append('0');
            }
            sb.append(stmp);
        }
        return sb.toString().toUpperCase();
    }
    public static void main(String[] args) {
        String productKey = "The productKey of your device";
        String deviceName = "The deviceName of your device";
        String deviceSecret = "The deviceSecret of your device";
        IotHttpClient client = new IotHttpClient();
        client.conenct(productKey, deviceName, deviceSecret);
        // The topic to which messages are sent. In the IoT Platform console, you can creat
e a custom topic and grant the publish permission to devices.
        String updateTopic = "/" + productKey + "/" + deviceName + "/user/update";
        client.publish(updateTopic, "hello http".getBytes(StandardCharsets.UTF_8));
        client.publish(updateTopic, new byte[] { 0x01, 0x02, 0x03 });
    }
}
```

1.4. Access IoT Platform by using MQTT over WebSocket connections

This topic describes how to connect a device to IoT Platform by using Message Queuing Telemetry Transport (MQTT) protocol over WebSocket connections. This topic also provides the corresponding sample code for Node.js.

Context

For more information about how to connect a device to IoT platform by using the WebSocket protocol, see Establish MQTT over WebSocket connections.

This topic describes how to use the device Link SDK provided by IoT Platform to connect a device and perform upstream and downstream messaging. A device of a previous public instance is used in the following example.

> **Note** The device Link SDK integrates the Transport Layer Security (TLS) encryption feature. No manual configurations are required.

Procedure

1. Download and install Node.js on the Windows or Linux operating system. In this example, Windows 10 (64-bit) is used. Download the node-v14.15.1-x64.msi installation package.
2. Open a Command Prompt window and run the following command to view the node version:

   ```
   node --version
   ```

   If the package is decompressed, the following version number appears:

   ```
   v14.15.1
   ```

3. Create a JavaScript file such as *iot_device.js* on your computer to store the Node.js sample code.

   The following Node.js sample code is used:

```
const iot = require('alibabacloud-iot-device-sdk');
// Specify the device certificate information.
const productKey = 'a1W***';
const deviceName = 'device2';
const deviceSecret = 'ff01e59d1a***';
// Specify the instance ID for a new public instance or Enterprise Edition instance. Yo
u do not need to specify this parameter if you are using a previous public instance.
const instanceId = '';
// Specify the ID of the region to which the product and device belong.
const region = 'cn-shanghai';
const brokerUrl = instanceId
  ? `wss://${instanceId}.mqtt.iothub.aliyuncs.com:443`
  : `wss://${productKey}.iot-as-mqtt.${region}.aliyuncs.com:443`;
const device = iot.device({
  productKey: `${productKey}`,
  deviceName: `${deviceName}`,
  deviceSecret: `${deviceSecret}`,
  brokerUrl,
  tls: true,
});
// Listen to connect events: Connect the device to IoT Platform over an MQTT connection
, enable the device to subscribe to a custom topic, and use the custom topic to send me
ssages to IoT Platform.
device.on('connect', () => {
  device.subscribe(`/${productKey}/${deviceName}/user/get`);
  console.log('connect successfully!');
  device.publish(`/${productKey}/${deviceName}/user/update`, 'hello world!');
});
// Listen to message events.
device.on('message', (topic, payload) => {
  console.log(topic, payload.toString());
});
// Listen to error events.
device.on('error', (error) => {
  console.error(error);
});
// To disconnect the device from IoT Platform, delete the double slashes (//) at the be
ginning of the following code, and call the end function.
//device.end();
```

You must specify parameters in the following table based on the actual use case such as the device information.

Parameter	Example	Description
productKey	a1W***	The device certificate that is saved after you add the device. For more information, see Create a device. You can obtain the certificate information of **device2** on the **Device Details** page in the IoT Platform console.
deviceName	device2	
deviceSecret	ff01e59d1a***	

Parameter	Example	Description
instanceId	''	The ID of the instance. You can view the ID of the instance on the **Overview** page in the IoT Platform console. ○ If you have an ID value, you must specify the ID for this parameter. ○ If no **Overview** or ID is generated for your instance, specify an empty string (`iotInstanceId = ''`) for the parameter.
region	cn-shanghai	The ID of the region where your IoT Platform instance resides. For more information about region IDs, see Regions and zones.

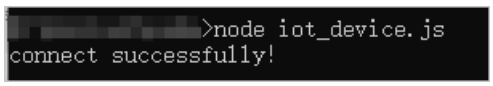
4. Open Command Prompt and run the cd command to go to the directory where the *iot_device.js* file resides. In this directory, run the npm command to download the device Link SDK library. The following figure shows the downloaded library.

```
npm install alibabacloud-iot-device-sdk --save
```

5. In Command Prompt, enter the following command to run the *iot_device.js* file. This way, the device is started.

```
node iot_device.js
```

The following response indicates that the device is connected to IoT Platform and messages are published.



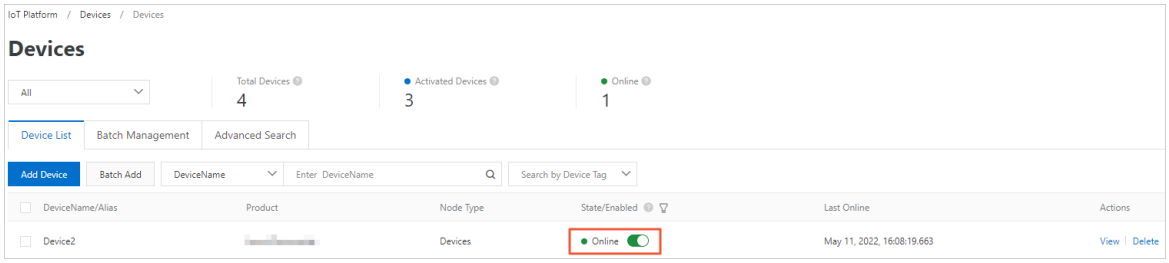
View logs and test downstream messaging

1. Log on to the IoT Platform console.
2. In the upper-left corner of the IoT Platform console, select the region where the IoT Platform instance resides.
3. On the **Instance Overview** page, find the instance and click the instance name to go to the **Instance Details** page.

> **Notice** IoT Platform provides instances only in the Japan (Tokyo) region. For other regions, please skip this step.

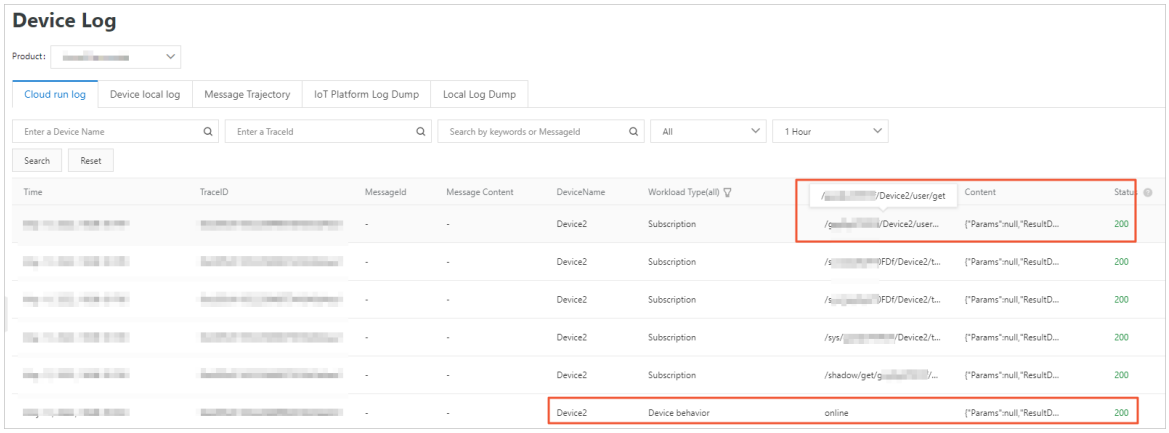
4. In the left-side navigation pane, choose **Devices > Devices**.

device2 is displayed on the **Device List** tab, and the device is in the **Online** state.

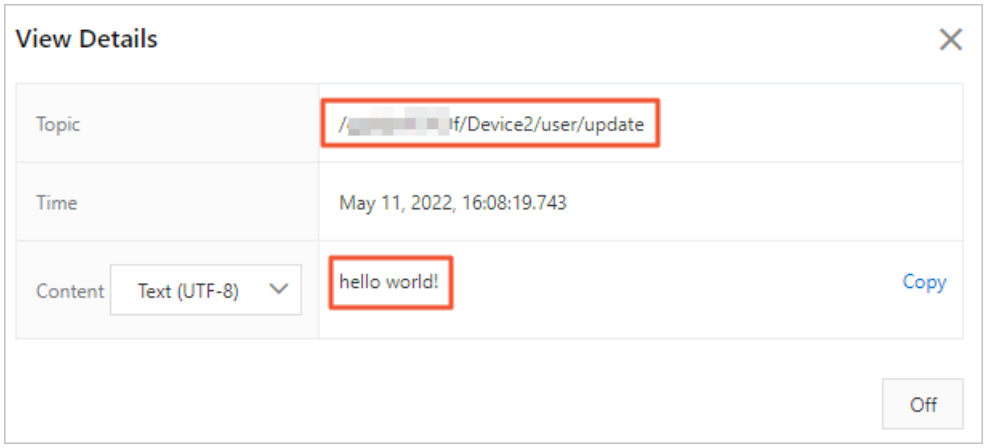


5. In the Actions column of **device2**, click **View**. On the **Device Details** page, click the **Device Log** tab, and then click **View**.

   View the device logs on the **Cloud run log** tab of the Device Log page.

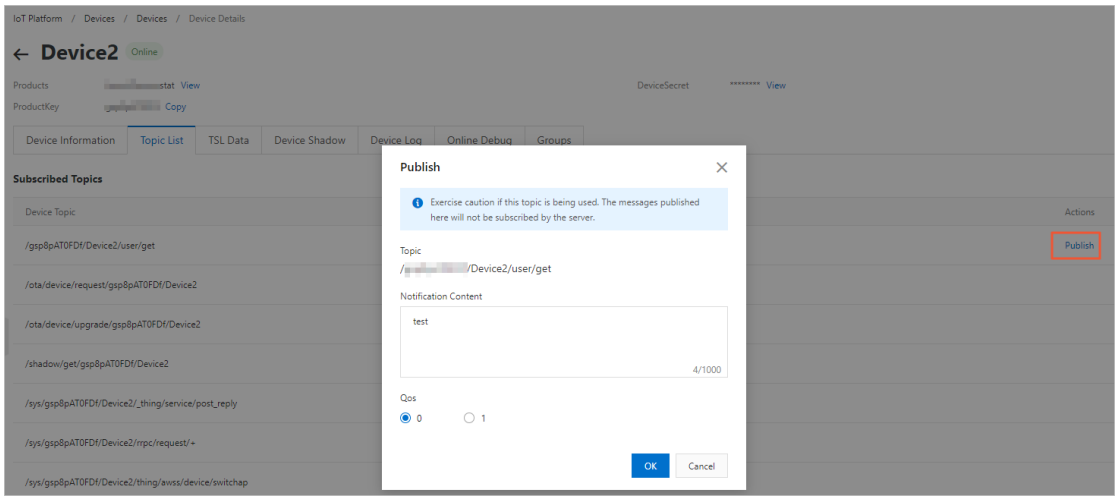


6. In the log list, find the message whose Workload Type is **Device-to-cloud Messages** and click **View** in the Message Content column to view the message content.



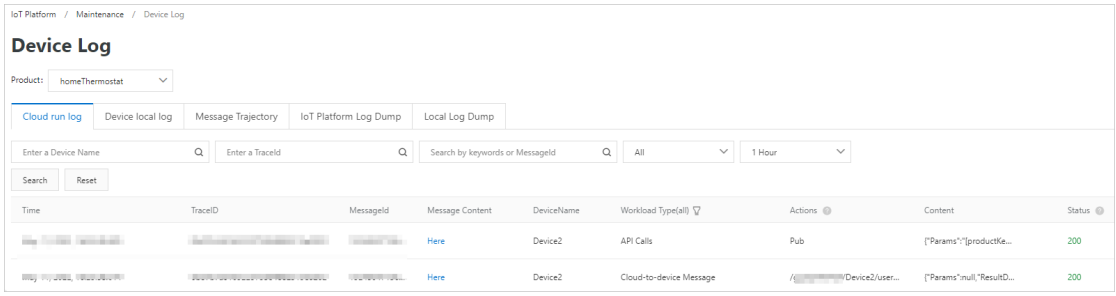
7. Test downstream messaging by sending a message from IoT Platform to the device.
   i. Choose **Devices > Devices**. On the **Device List** tab of the Devices page, find **device2** and click **View** in the Actions column.
   ii. On the **Device Details** page, click the **Topic List** tab, find the `/a1W***/device2/user/get` topic in Subscribed Topics, and click **Publish**.

iii. Enter a message and click **OK**.



iv. Return to Command Prompt and check whether the device has received the message. If downstream messaging works properly, the message is received.

You can also return to the **Cloud run log** tab to view messaging logs.



1.5. MQTT-based dynamic registration

This article describes how to dynamically register the devices that use the MQTT protocol and obtain the DeviceSecrets. The DeviceSecrets are required for authentication when you connect the devices to IoT Platform. In this example, the sample Java code is used for pre-registration unique-certificate-per-product authentication.

Prerequisites

The following steps that are specified in the Unique-certificate-per-product authentication topic are performed:

Background information

IoT Platform supports multiple authentication methods for devices. For more information, see Authenticate a device.

You can establish MQTT connections to perform pre-registration unique-certificate-per-product authentication or preregistration-free unique-certificate-per-product authentication. For more information about the procedure and parameters of MQTT-based dynamic registration, see MQTT-based dynamic registration.

Procedure

1. Open IntelliJ IDEA and create a Maven project. In this example, the **MQTT dynamic registration** project is created.
2. In the pom.xml file, add the following Maven dependencies and click Load Maven Changes to download the packages.

```
<dependency>
  <groupId>org.eclipse.paho</groupId>
  <artifactId>org.eclipse.paho.client.mqttv3</artifactId>
  <version>1.2.1</version>
</dependency>
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>fastjson</artifactId>
  <version>1.2.61</version>
</dependency>
```

3. Go to the *\MQTT dynamic registration \src\main\java* directory and create a Java class. In this example, the DynamicRegisterByMqtt class is created. Enter the following code.

> **Note**
> - If the device is not activated, dynamic registration can be performed multiple times. In this case, only the latest DeviceSecret is valid. Make sure that the latest DeviceSecret is persisted on the device.
> - If the device is activated, you must call the ResetThing operation to reset the device status in IoT Platform to inactive. Then, you can dynamically register the device.

```
/*
 * Copyright © 2019 Alibaba. All rights reserved.
 */
package com.aliyun.iot.demo;
import java.nio.charset.StandardCharsets;
import java.util.Random;
import java.util.Set;
import java.util.SortedMap;
import java.util.TreeMap;
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttCallback;
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.eclipse.paho.client.mqttv3.MqttMessage;
import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence;
import com.alibaba.fastjson.JSONObject;
/**
 * Perform dynamic registration for a device.
 */
public class DynamicRegisterByMqtt {
    // The ID of the region where your product resides.
```

```
    private static String regionId = "cn-shanghai";
    // Specify an encryption algorithm. Valid values: hmacmd5, hmacsha1, and hmacsha256
. The value that you specify must be the same as the value of the signmethod parameter.
    private static final String HMAC_ALGORITHM = "hmacsha1";
    // The topic that receives device certificates from IoT Platform. Use the topic wit
hout any changes. You do not need to create or subscribe to the topic.
    private static final String REGISTER_TOPIC = "/ext/register";
    /**
     * Dynamic registration.
     *
     * @ param productKey: the ProductKey of the product.
     * @param productSecret: the ProductSecret of the device.
     * @param deviceName: the DeviceName of the device.
     * @throws Exception
     */
    public void register(String productKey, String productSecret, String deviceName) th
rows Exception {
        // The endpoint of the authentication service. You must use the Transport Layer
Security (TLS) protocol.
        String broker = "ssl://" + productKey + ".iot-as-mqtt." + regionId + ".aliyuncs
.com:1883";
        // The client ID. We recommend that you use the MAC address or SN of the device
. The client ID can be a maximum of 64 characters in length.
        String clientId = productKey + "." + deviceName;
        // Obtain a random value.
        Random r = new Random();
        int random = r.nextInt(1000000);
        // The value of the securemode parameter is 2 and cannot be changed. The value
2 indicates that TLS is applied. The signmethod parameter specifies an encryption algor
ithm.
        String clientOpts = "|securemode=2,authType=register,signmethod=" + HMAC_ALGORI
THM + ",random=" + random + "|";
        // The ID of the MQTT client.
        String mqttClientId = clientId + clientOpts;
        // The usename of the MQTT client.
        String mqttUsername = deviceName + "&" + productKey;
        // The signature that the MQTT client uses to connect to the endpoint.
        JSONObject params = new JSONObject();
        params.put("productKey", productKey);
        params.put("deviceName", deviceName);
        params.put("random", random);
        String mqttPassword = sign(params, productSecret);
        // Send an MQTT CONNECT message for dynamic registration.
        connect(broker, mqttClientId, mqttUsername, mqttPassword);
    }
    /**
     * Send an MQTT CONNECT message for dynamic registration.
     *
     * @param serverURL: the endpoint for dynamic registration.
     * @param clientId: the ID of the client.
     * @param username: the username of the MQTT client.
     * @param password: the password of the MQTT client.
     */
    @SuppressWarnings("resource")
```

```
    private void connect(String serverURL, String clientId, String username, String pas
sword) {
        try {
            MemoryPersistence persistence = new MemoryPersistence();
            MqttClient sampleClient = new MqttClient(serverURL, clientId, persistence);
            MqttConnectOptions connOpts = new MqttConnectOptions();
            connOpts.setMqttVersion(4);// MQTT 3.1.1
            connOpts.setUserName(username);// The username.
            connOpts.setPassword(password.toCharArray());// The password.
            connOpts.setAutomaticReconnect(false); // Disable the automatic reconnectio
n feature based on MQTT rules that are set for dynamic registration.
            System.out.println("----- register params -----");
            System.out.print("server=" + serverURL + ",clientId=" + clientId);
            System.out.println(",username=" + username + ",password=" + password);
            sampleClient.setCallback(new MqttCallback() {
                @Override
                public void messageArrived(String topic, MqttMessage message) throws Ex
ception {
                    // Print out only the response of dynamic registration.
                    if (REGISTER_TOPIC.equals(topic)) {
                        String payload = new String(message.getPayload(), StandardChars
ets.UTF_8);
                        System.out.println("----- register result -----");
                        System.out.println(payload);
                        sampleClient.disconnect();
                    }
                }
                @Override
                public void deliveryComplete(IMqttDeliveryToken token) {
                }
                @Override
                public void connectionLost(Throwable cause) {
                }
            });
            sampleClient.connect(connOpts);
        } catch (MqttException e) {
            System.out.print("register failed: clientId=" + clientId);
            System.out.println(",username=" + username + ",password=" + password);
            System.out.println("reason " + e.getReasonCode());
            System.out.println("msg " + e.getMessage());
            System.out.println("loc " + e.getLocalizedMessage());
            System.out.println("cause " + e.getCause());
            System.out.println("excep " + e);
            e.printStackTrace();
        }
    }
    /**
     * Generate a signature for dynamic registration.
     *
     * @param params: the required parameters that you can use to generate a signature.
     * @param productSecret: the ProductSecret of the device.
     * @return: a hexadecimal signature string.
     */
    private String sign(JSONObject params, String productSecret) {
```

```
        // Sort request parameters in alphabetical order.
        Set<String> keys = getSortedKeys(params);
        // Remove the sign and signMethod parameters.
        keys.remove("sign");
        keys.remove("signMethod");
        // Obtain the plaintext of the signature.
        StringBuffer content = new StringBuffer();
        for (String key : keys) {
            content.append(key);
            content.append(params.getString(key));
        }
        // Generate a signature.
        String sign = encrypt(content.toString(), productSecret);
        System.out.println("sign content=" + content);
        System.out.println("sign result=" + sign);
        return sign;
    }
    /**
     * Convert a JSON object to a set of key-value pairs.
     *
     * @param json: the JSON object to be converted.
     * @return: a set of key-value pairs that are converted from the JSON object.
     */
    private Set<String> getSortedKeys(JSONObject json) {
        SortedMap<String, String> map = new TreeMap<String, String>();
        for (String key : json.keySet()) {
            String vlaue = json.getString(key);
            map.put(key, vlaue);
        }
        return map.keySet();
    }
    /**
     * Specify an encryption algorithm for the HMAC_ALGORITHM parameter.
     *
     * @param content: the plaintext.
     * @param secret: the encryption key.
     * @return: the ciphertext.
     */
    private String encrypt(String content, String secret) {
        try {
            byte[] text = content.getBytes(StandardCharsets.UTF_8);
            byte[] key = secret.getBytes(StandardCharsets.UTF_8);
            SecretKeySpec secretKey = new SecretKeySpec(key, HMAC_ALGORITHM);
            Mac mac = Mac.getInstance(secretKey.getAlgorithm());
            mac.init(secretKey);
            return byte2hex(mac.doFinal(text));
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
    /**
     * Convert a binary array to a hexadecimal string.
     *
     * @param b: the binary array
```

```
     * @param b: the binary array.
     * @return: a hexadecimal string.
     */
    private String byte2hex(byte[] b) {
        StringBuffer sb = new StringBuffer();
        for (int n = 0; b != null && n < b.length; n++) {
            String stmp = Integer.toHexString(b[n] & 0XFF);
            if (stmp.length() == 1) {
                sb.append('0');
            }
            sb.append(stmp);
        }
        return sb.toString().toUpperCase();
    }
    public static void main(String[] args) throws Exception {
        String productKey = "a1IoK******";
        String productSecret = "6vEu5Qlj5S******";
        String deviceName = "OvenDevice01";
        // Perform dynamic registration.
        DynamicRegisterByMqtt client = new DynamicRegisterByMqtt();
        client.register(productKey, productSecret, deviceName);
        // After dynamic registration is successful, persist the DeviceSecret on premis
es.
    }
}
```

Parameter	Example	Description
regionId	cn-shanghai	
productKey	a1IoK******	The ProductKey that is burned to the device. You can log on to the IoT Platform console and view the ProductKey on the **Product Details** page.
productSecret	6vEu5Qlj5S******	The ProductSecret that is burned to the device. You can log on to the IoT Platform console and view the ProductSecret on the **Product Details** page.
deviceName	OvenDevice01	The name of the device.

4. Run the DynamicRegisterByMqtt.java file so that the device can send an authentication request to IoT Platform. The DeviceName, ProductKey, and ProductSecret are included in the request.

   The following figure shows the result. After the device passes the authentication, the device receives a DeviceSecret that is issued by IoT Platform. In this example, the DeviceSecret is `8d1f0cdab49dd229cf3b75**********`.

What to do next

After the device obtains the device certificate (ProductKey, DeviceName, and DeviceSecret), you can use the MQTT client to connect the device with IoT Platform for data communication.

For more information, see Use the Paho MQTT Java client.

1.6. HTTPS-based dynamic registration

This article describes how to dynamically register the devices that use the HTTPS protocol and obtain the DeviceSecrets. The DeviceSecrets are required for authentication when you connect the devices to IoT Platform. In this example, the sample code for Java is used.

Prerequisites

The following steps that are specified in the Unique-certificate-per-product authentication topic are performed:

Background information

IoT Platform supports multiple authentication methods for devices. For more information, see Authenticate a device.

You can establish HTTPS connections to perform pre-registration unique-certificate-per-product authentication. For more information about the topics and parameters that are used for HTTPS-based dynamic registration, see HTTPS-based dynamic registration of directly connected devices.

Procedure

1. Open IntelliJ IDEA and create a Maven project. In this example, the **HTTPS dynamic registration** project is created.
2. In the pom.xml file, add the following Maven dependencies and click Load Maven Changes to download the packages.

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>fastjson</artifactId>
  <version>1.2.61</version>
</dependency>
```

3. Go to the *\HTTPS dynamic registration \src\main\java* directory and create a Java class. In this example, the DynamicRegisterByMqtt class is created. Enter the following code.

> **Note**
>
> - If the device is not activated, dynamic registration can be performed multiple times. In this case, only the latest DeviceSecret is valid. Make sure that the latest DeviceSecret is persisted on the device.
> - If the device is activated, you must call the ResetThing operation to reset the device status in IoT Platform to inactive. Then, you can dynamically register the device.

```
/*
 * Copyright © 2019 Alibaba. All rights reserved.
 */
```

```
package com.aliyun.iot.demo;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.URL;
import java.nio.charset.StandardCharsets;
import java.util.Random;
import java.util.Set;
import java.util.SortedMap;
import java.util.TreeMap;
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import javax.net.ssl.HttpsURLConnection;
import com.alibaba.fastjson.JSONObject;
/**
 * Perform dynamic registration for a device.
 */
public class DynamicRegisterByHttps {
    // The ID of the region where your product resides.
    private static String regionId = "cn-shanghai";
    // Specify an encryption algorithm. Valid values: hmacmd5, hmacsha1, and hmacsha256
. The value that you specify must be the same as the value of the signmethod parameter.
    private static final String HMAC_ALGORITHM = "hmacsha1";
    /**
     * Dynamic registration.
     *
     * @param productKey: the ProductKey of the device.
     * @param productSecret: the ProductSecret of the device.
     * @param deviceName: the DeviceName of the device.
     * @throws Exception
     */
    public void register(String productKey, String productSecret, String deviceName) th
rows Exception {
        // The requested URL.
        URL url = new URL("https://iot-auth." + regionId + ".aliyuncs.com/auth/register
/device");
        HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
        conn.setRequestMethod("POST");
        conn.setRequestProperty("Content-type", "application/x-www-form-urlencoded");
        conn.setDoOutput(true);
        conn.setDoInput(true);
        // Obtain the output stream of the URLConnection object.
        PrintWriter out = new PrintWriter(conn.getOutputStream());
        // Send request parameters.
        out.print(registerdBody(productKey, productSecret, deviceName));
        // Clear the content of the cache.
        out.flush();
        // Obtain the input stream of the URLConnection object.
        BufferedReader in = new BufferedReader(new InputStreamReader(conn.getInputStrea
m()));
        // Obtain a response from the endpoint.
        String result = "";
        String line = "";
        while ((line = in.readLine()) != null) {
```

```
            result += line;
        }
        System.out.println("----- register result -----");
        System.out.println(result);
        // Close input and output streams.
        in.close();
        out.close();
        conn.disconnect();
    }
    /**
     * Generate a request for dynamic registration.
     *
     * @param productKey: the ProductKey of the device.
     * @param productSecret: the ProductSecret of the device.
     * @param deviceName: the DeviceName of the device.
     * @return: the response payloads.
     */
    private String registerdBody(String productKey, String productSecret, String device
Name) {
        // Obtain a random value.
        Random r = new Random();
        int random = r.nextInt(1000000);
        // The required parameters for dynamic registration.
        JSONObject params = new JSONObject();
        params.put("productKey", productKey);
        params.put("deviceName", deviceName);
        params.put("random", random);
        params.put("signMethod", HMAC_ALGORITHM);
        params.put("sign", sign(params, productSecret));
        // Concatenate the payloads.
        StringBuffer payload = new StringBuffer();
        for (String key : params.keySet()) {
            payload.append(key);
            payload.append("=");
            payload.append(params.getString(key));
            payload.append("&");
        }
        payload.deleteCharAt(payload.length() - 1);
        System.out.println("----- register payload -----");
        System.out.println(payload);
        return payload.toString();
    }
    /**
     * Generate a signature for dynamic registration.
     *
     * @param params: the required parameters that you can use to generate a signature.
     * @param productSecret: the ProductSecret of the device.
     * @return: the signature in the hexadecimal format.
     */
    private String sign(JSONObject params, String productSecret) {
        // Sort request parameters in alphabetical order.
        Set<String> keys = getSortedKeys(params);
        // Remove the sign and signMethod parameters
        keys.remove("sign");
```

```
        keys.remove("signMethod");
        // Obtain the plaintext of the signature.
        StringBuffer content = new StringBuffer();
        for (String key : keys) {
            content.append(key);
            content.append(params.getString(key));
        }
        // Generate a signature.
        String sign = encrypt(content.toString(), productSecret);
        System.out.println("sign content=" + content);
        System.out.println("sign result=" + sign);
        return sign;
    }
    /**
     * Convert a JSON object to a set of key-value pairs.
     *
     * @param json: the JSON object to be converted.
     * @return: a set of key-value pairs that are converted from the JSON object.
     */
    private Set<String> getSortedKeys(JSONObject json) {
        SortedMap<String, String> map = new TreeMap<String, String>();
        for (String key : json.keySet()) {
            String vlaue = json.getString(key);
            map.put(key, vlaue);
        }
        return map.keySet();
    }
    /**
     * Specify an encryption algorithm for the HMAC_ALGORITHM parameter.
     *
     * @param content: the plaintext.
     * @param secret: the encryption key.
     * @return: the ciphertext.
     */
    private String encrypt(String content, String secret) {
        try {
            byte[] text = content.getBytes(StandardCharsets.UTF_8);
            byte[] key = secret.getBytes(StandardCharsets.UTF_8);
            SecretKeySpec secretKey = new SecretKeySpec(key, HMAC_ALGORITHM);
            Mac mac = Mac.getInstance(secretKey.getAlgorithm());
            mac.init(secretKey);
            return byte2hex(mac.doFinal(text));
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
    /**
     * Convert a binary array to a hexadecimal string.
     *
     * @param b: the binary array.
     * @return: a hexadecimal string.
     */
    private String byte2hex(byte[] b) {
```

```
        StringBuffer sb = new StringBuffer();
        for (int n = 0; b != null && n < b.length; n++) {
            String stmp = Integer.toHexString(b[n] & 0XFF);
            if (stmp.length() == 1) {
                sb.append('0');
            }
            sb.append(stmp);
        }
        return sb.toString().toUpperCase();
    }
    public static void main(String[] args) throws Exception {
        String productKey = "a1IoK******";
        String productSecret = "6vEu5Qlj5S******";
        String deviceName = "OvenDevice01";
        // Perform dynamic registration.
        DynamicRegisterByHttps client = new DynamicRegisterByHttps();
        client.register(productKey, productSecret, deviceName);
        // After dynamic registration is successful, persist the DeviceSecret on premis
es.
    }
}
```

Parameter	Example	Description
regionId	cn-shanghai	
productKey	a1IoK******	The ProductKey that is burned to the device. You can log on to the IoT Platform console and view the ProductKey on the **Product Details** page.
productSecret	6vEu5Qlj5S******	The ProductSecret that is burned to the device. You can log on to the IoT Platform console and view the ProductSecret on the **Product Details** page.
deviceName	OvenDevice01	The name of the device.

4. Run the DynamicRegisterByMqtt.java file so that the device can send an authentication request to IoT Platform. The DeviceName, ProductKey, and ProductSecret are included in the request. The following figure shows the result. After the device passes the authentication, the device receives a DeviceSecret that is issued by IoT Platform. In this example, the DeviceSecret is `6b14088fa377e8f852d82f7f********`.

What to do next

After the device obtains the device certificate (ProductKey, DeviceName, and DeviceSecret), you can use the MQTT client to connect the device with IoT Platform for data communication.

For more information, see Use the Paho MQTT Java client.

1.7. Connect a device to IoT Platform by using MQTT.fx

MQTT.fx is an Eclipse Paho-based Message Queuing Telemetry Transport (MQTT) client that is written in Java. MQTT.fx supports Windows, macOS, and Linux operating systems. MQTT.fx can be used to verify whether a device can connect to IoT Platform. MQTT.fx allows you to subscribe to and publish messages by using topics. This article describes how to connect a simulated device to IoT Platform over MQTT by using MQTT.fx on Windows.

Prerequisites

A product and a device are created in the IoT Platform console. The device certificate and **MQTT connection parameters** are obtained from the **Device Details** page. For more information, see the following articles:

- Create a product
- Create a device
- How do I obtain MQTT parameters for authentication?

In this example, a public instance of the previous version is used. The following table describes the device certificate and MQTT connection parameters. For more information about the parameters, see Establish MQTT connections over TCP.

Parameter	Value
ProductKey	`a1***`
DeviceName	device1
DeviceSecret	f35***d9e
clientId	`a1***.device1securemode=2,signmethod=hmacsha256,timestamp=2524608000000`
username	`device1&a1***`
passwd	`86761***21d`
mqttHostUrl	`a1***.iot-as-mqtt.cn-shanghai.aliyuncs.com`
port	`1883`

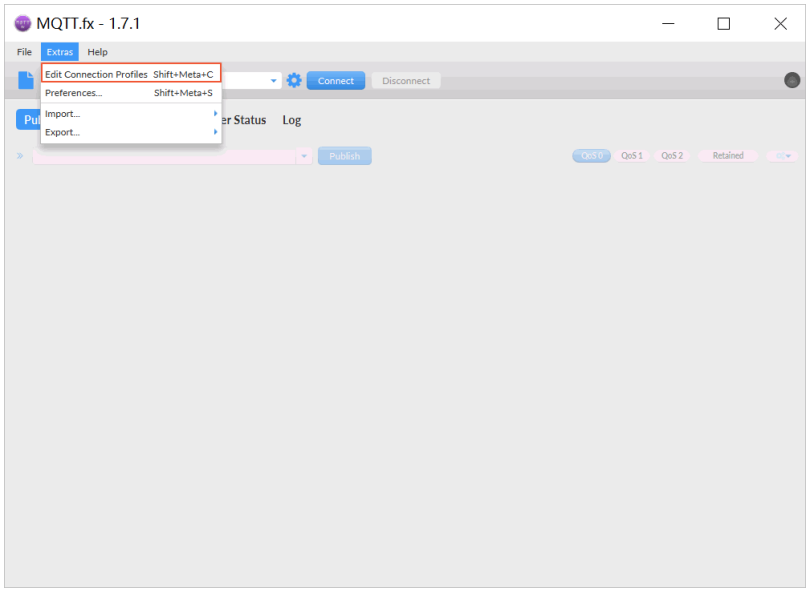
Notice MQTT.fx is used to simulate an online device and allows you to transmit only non-passthrough data. To transmit pass-through data, you can use an actual device or an SDK for testing.

Configure MQTT.fx

1. Download and install the MQTT.fx tool.

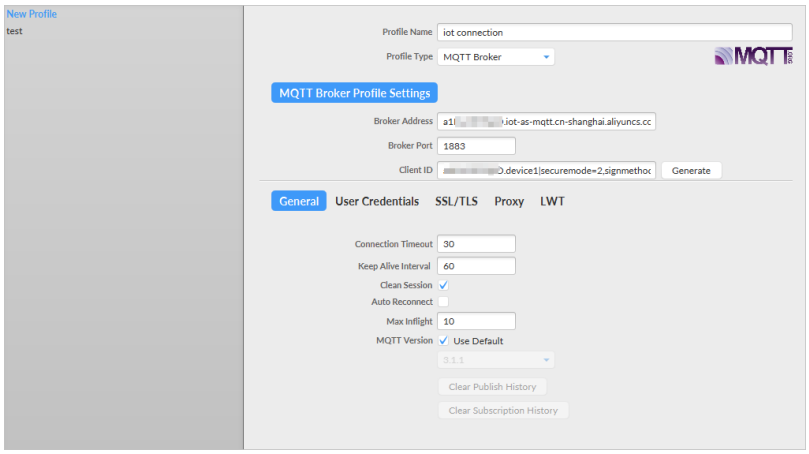
   For more information, visit the MQTT.fx.

2. Start the MQTT.fx tool, click **Extras** in the menu bar, and then select **Edit Connection Profiles**.



3. On the **Edit Connection Profiles** page, configure the parameters.

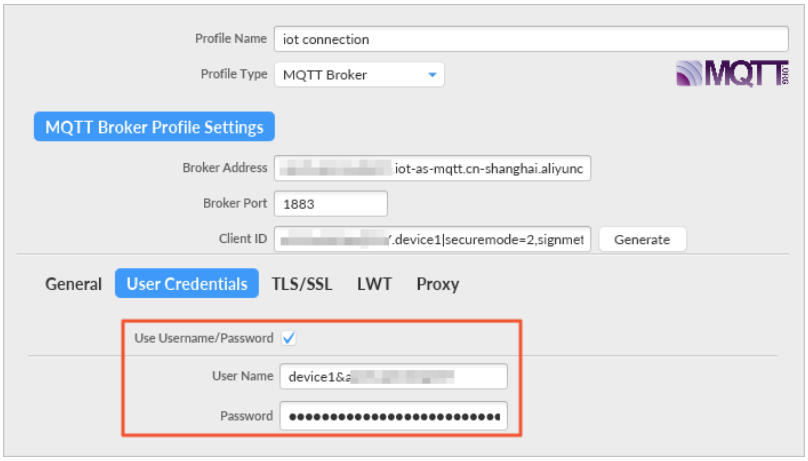
   i. Specify the basic information.



Parameter	Description
Profile Name	Enter a custom name. In this example, *iot connection* is specified.
Profile Type	Specify a connection mode. Select **MQTT Broker**.
MQTT Broker Profile Settings	
Broker Address	Enter an MQTT endpoint, which is the value of the mqttHostUrl parameter that you obtained in the "*Prerequisites*" section of this article. In this example, *a1***.iot-as-mqtt.cn-shanghai.aliyuncs.com* is specified. ■ *a1**** indicates the ProductKey. ■ *cn-shanghai* indicates the region ID.
Broker Port	Enter a port number. Set this parameter to *1883*.

Parameter	Description
Client ID	Configure the parameters in the MQTT protocol. Format: *${ClientId}*`securemode=`*${Mode}*`,signmethod=`*${SignMethod}*`timestamp=`*${timestamp}*. Enter the value of the clientId parameter that you obtained in the "*Prerequisites*" section of this article. In this example, *a1***.device1securemode=2,signmethod=hmacsha256,timestamp=2524608000000* is specified. The following table describes the parameters. ■ *${ClientId}*: the ID of the client, such as a device, an application, or a web browser. **Notice** You can specify a client ID as needed. The client ID must be 1 to 64 characters in length. If the client ID is a device ID, we recommend that you specify the MAC address or the serial number (SN) of the device as the value of this parameter. This way, you can identify your devices in an efficient manner. ■ *${Mode}*: the security mode. Valid values: 2 and 3. If you use a direct TLS connection, specify *securemode=2*. In this case, you must configure the **SSL/TLS** parameters. If you use a direct TCP connection, specify *securemode=3*. In this case, you do not need to configure the **SSL/TLS** parameters. ■ *${SignMethod}*: the signature algorithm. Valid values: *hmacmd256*, *hmacmd5*, and *hmacsha1*. ■ *${timestamp}*: the timestamp of the request. Unit: milliseconds. This parameter is optional. In the connection parameters provided by IoT Platform, the value of the *${ClientId}* parameter is a string in the *${ProductKey}* + '.' + *${DeviceName}* format. The default value of *${Mode}* is *2*, and the default value of *${SignMethod}* is *hmacsha256*. You can modify the parameters as needed. **Notice** ■ Do not confuse the **Client ID** parameter of the MQTT.fx client with the *${ClientId}* parameter of the device. ■ Do not omit the vertical bars () between and at the end of the parameters. ■ When you configure the parameters, make sure that you remove all spaces from the parameter values. ■ After you configure the **Client ID** parameter, do not click **Generate**.
General	In this example, the default values of the parameters are used. You can configure the parameters based on your business requirements.

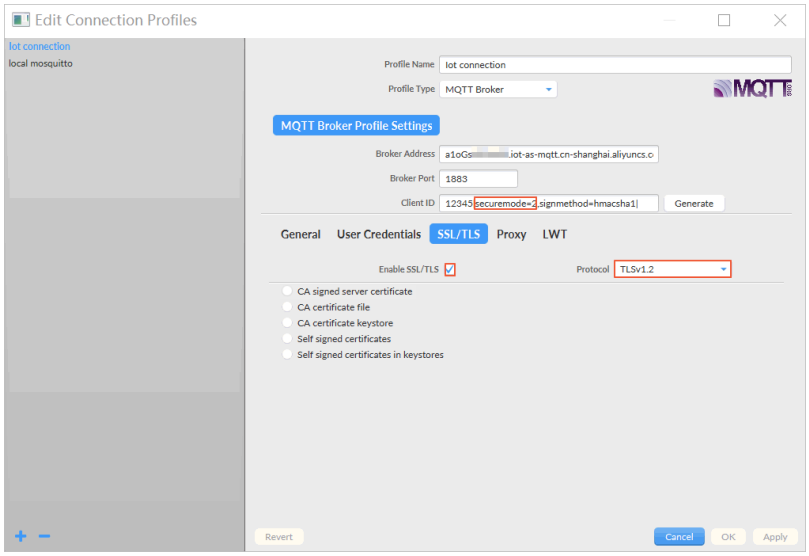
ii. Click the **User Credentials** tab. Configure the **User Name** and **Password** parameters.



Parameter	Description
User Name	The username consists of a DeviceName, an ampersand (&), and a ProductKey. Format: *${DeviceName}&${ProductKey}*. Enter the value of the username parameter that you obtained in the "*Prerequisites*" section of this article. In this example, *device1&a1**** is specified. ■ `device1` indicates the DeviceName of the device. ■ `a1***` indicates the ProductKey of the device.
Password	To generate a password, you must select a signature algorithm, use the DeviceSecret of the device as a secret key, and then concatenate the required parameters and the values of the parameters. Enter the value of the passwd parameter that you obtained in the "*Prerequisites*" section of this article. In this example, *86761***21d* is specified. **Notice** ■ Your MQTT.fx client may display a masked password. If a password is pasted, the pointer moves to the end of the password. In this case, you do not need to paste the password again. ■ Make sure that you specify valid uppercase and lowercase letters in parameter names and values. ■ You may modify the *${clientId}* and *${SignMethod}* parameters when you configure the **Client ID** parameter. In this case, make sure that the specified signature algorithm is the same as the value of the corresponding parameter in **Client ID**. Then, recalculate a password. For more information about the parameters and the calculation method, see Use the Node.js script.

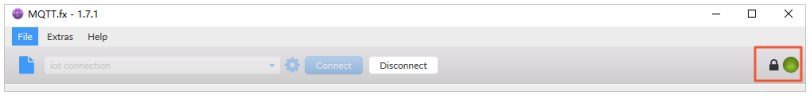
iii. If you use a TLS connection (securemode=2), click the **SSL/TLS** tab, select **Enable SSL/TLS**, and then set the **Protocol** parameter to **TLSv1.2**.

> **Notice** If you use a TCP connection (`securemode=3`), use the default settings on the SSL/TLS tab, and go to the next step.

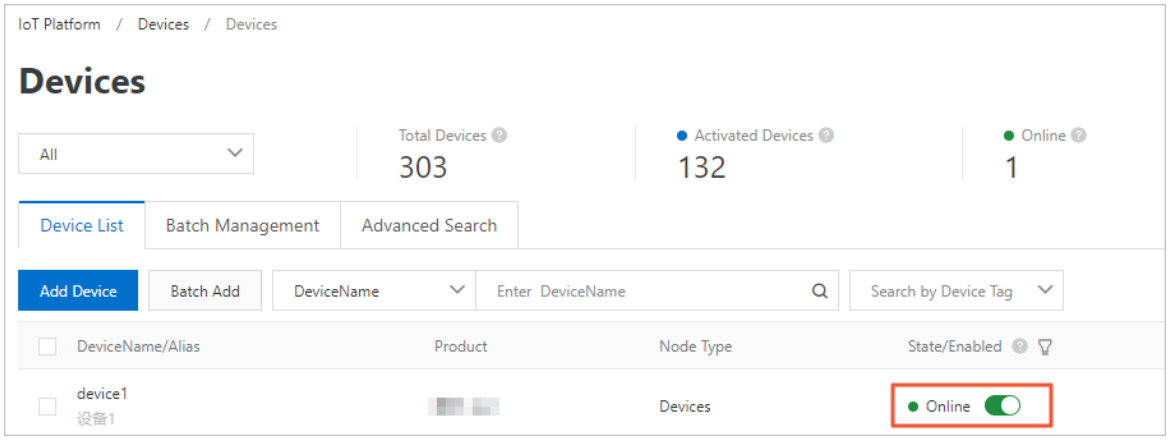


4. Click **OK** in the lower-right corner.
5. Click **Connect**.

   If the indicator on the right side turns green, the connection is established.



To view the status of the device, perform the following steps: Log on to the IoT Platform console. Choose **Devices > Devices**, select the product, and then find the device. The device is in the **Online** state.



In the following sections, downstream messaging and upstream messaging are tested to check whether the MQTT.fx client is connected to IoT Platform. If your test results are different from the following sample results, the connection is not established. You must modify the settings based on the logs.

Test downstream messaging

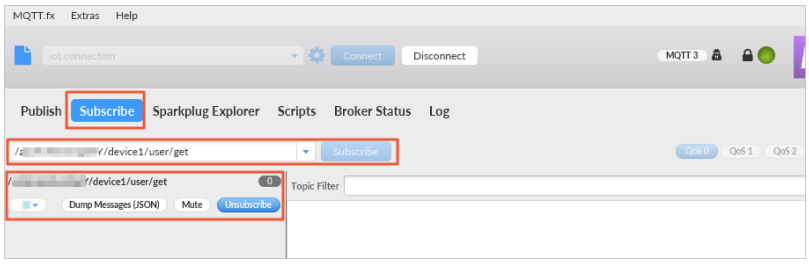
1. Log on to the IoT Platform console. On the Product Details page, choose **Topic Categories** > **Custom Topics**. Then, find a custom topic that has the **Subscribe** permission.

   In this example, the `/a1***/${deviceName}/user/get` topic is used. You must replace the `${deviceName}` variable with `device1`.

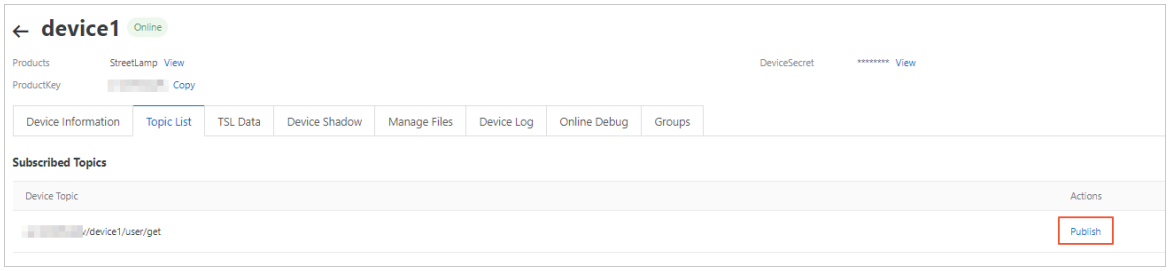
   For more information, see Custom topics.

2. In the MQTT.fx tool, click **Subscribe**. In the **Subscribe** field, enter the topic that you specified in the previous step, and then click **Subscribe**.

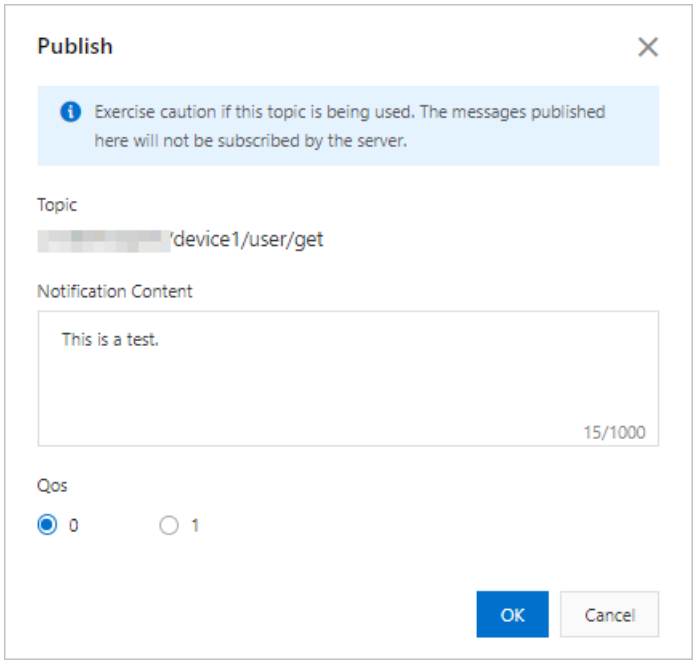
   If the subscription is successful, the custom topic appears on the Subscribe tab.



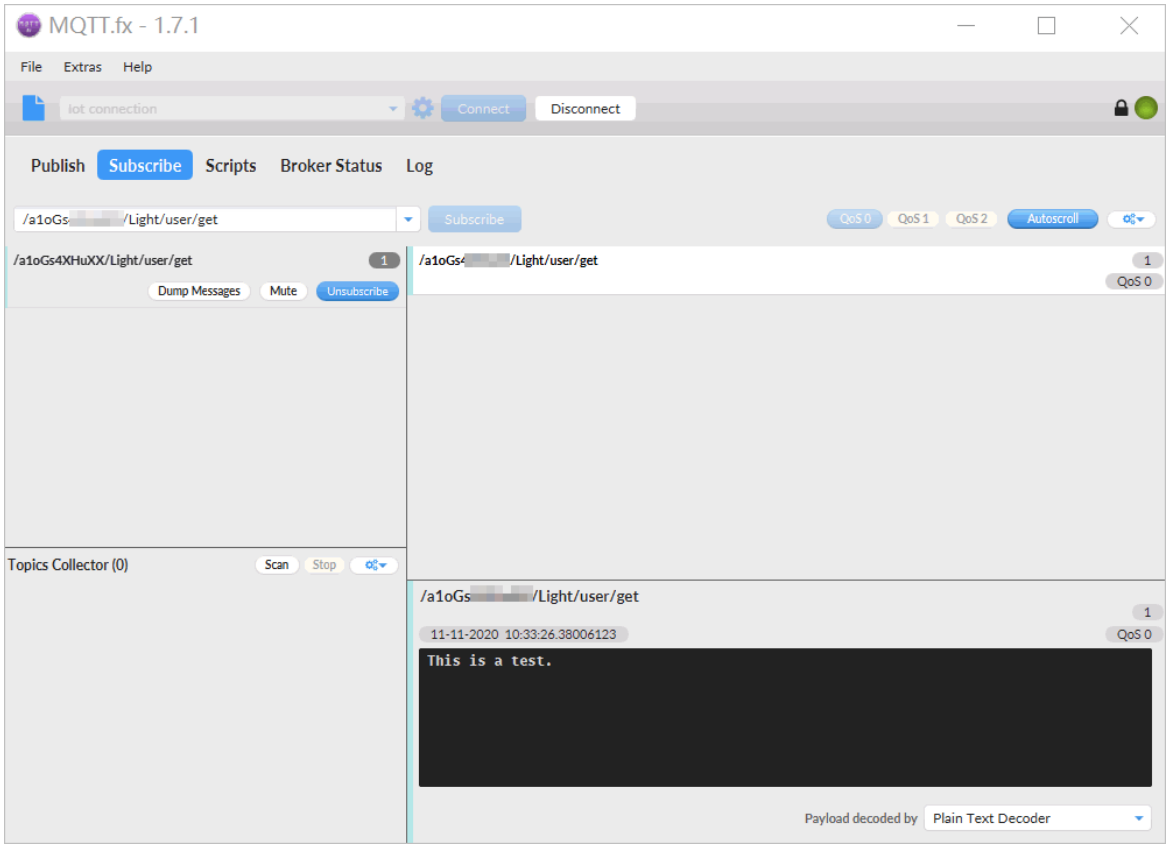
3. Go to the **Device Details** page in the IoT Platform console. On the **Topic List** tab, find the topic and click **Publish Message**.



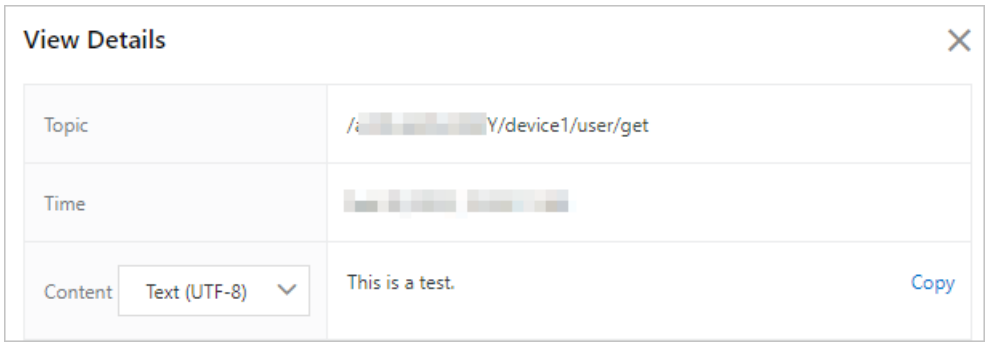
4. Enter a message and click **OK**.



5. In the MQTT.fx tool, check whether the message is received.



6. Go to the **Device Details** page in the IoT Platform console. On the **Device Log** tab, click **View**. On the **Device Log** page, view **cloud-to-device** messages.



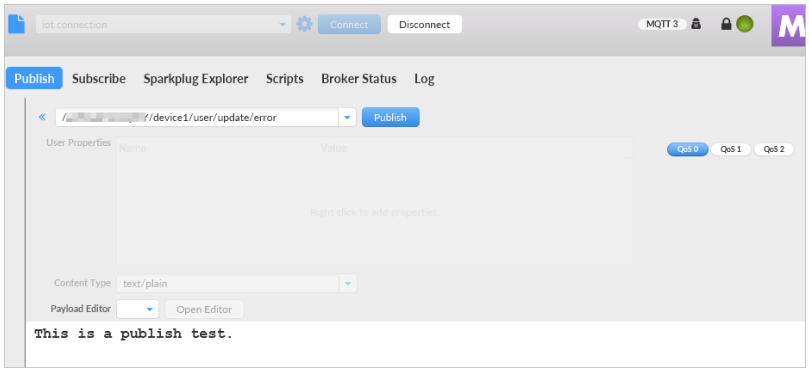
Test upstream messaging

1. Log on to the IoT Platform console. On the Product Details page, choose **Topic Categories** > **Custom Topics**. Then, find a custom topic that has the **Subscribe** permission.

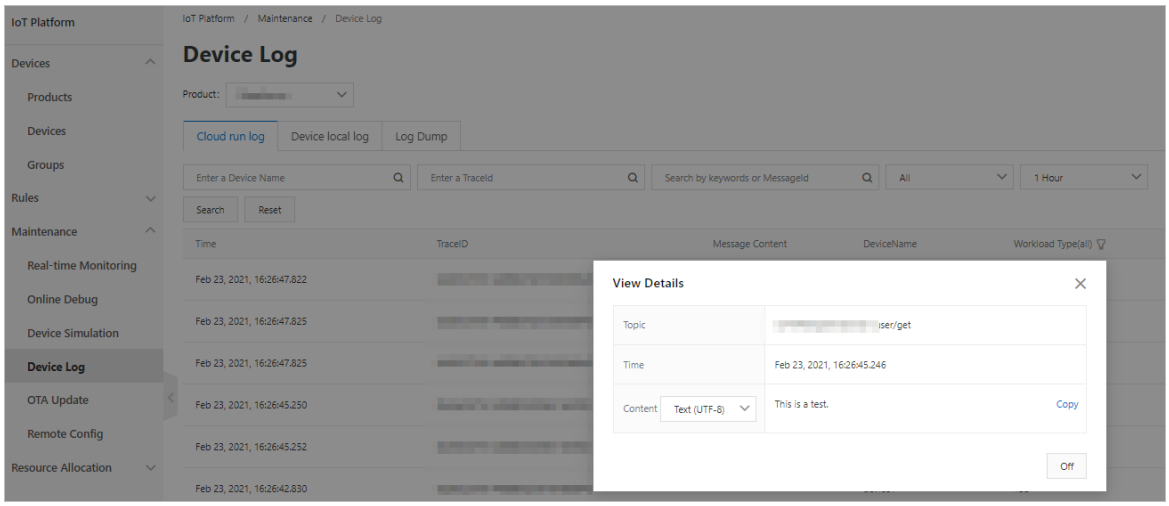
   In this example, the `/a1***/${deviceName}/user/get` topic is used. You must replace the `${deviceName}` variable with `device1`.

   For more information, see Custom topics.

2. In the MQTT.fx tool, click **Publish**. In the **Publish** field, enter the topic that you specified in the previous step. In the editor, enter the message to be sent and click **Publish**.

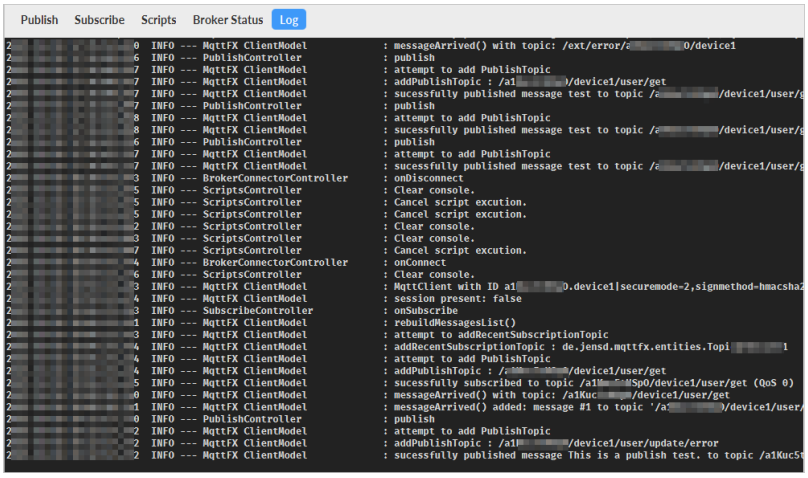


3. Go to the **Device Details** page in the IoT Platform console. On the **Device Log** tab, click **View**. On the **Device Log** page, view device-to-cloud messages.



View logs

In the MQTT.fx tool, click the **Log** tab. On the tab that appears, view the operation logs and error logs.



1.8. Connect Android Things hardware to IoT Platform

This article describes how to connect Google Android Things hardware to IoT Platform. An indoor air test project is used in this example.

Hardware

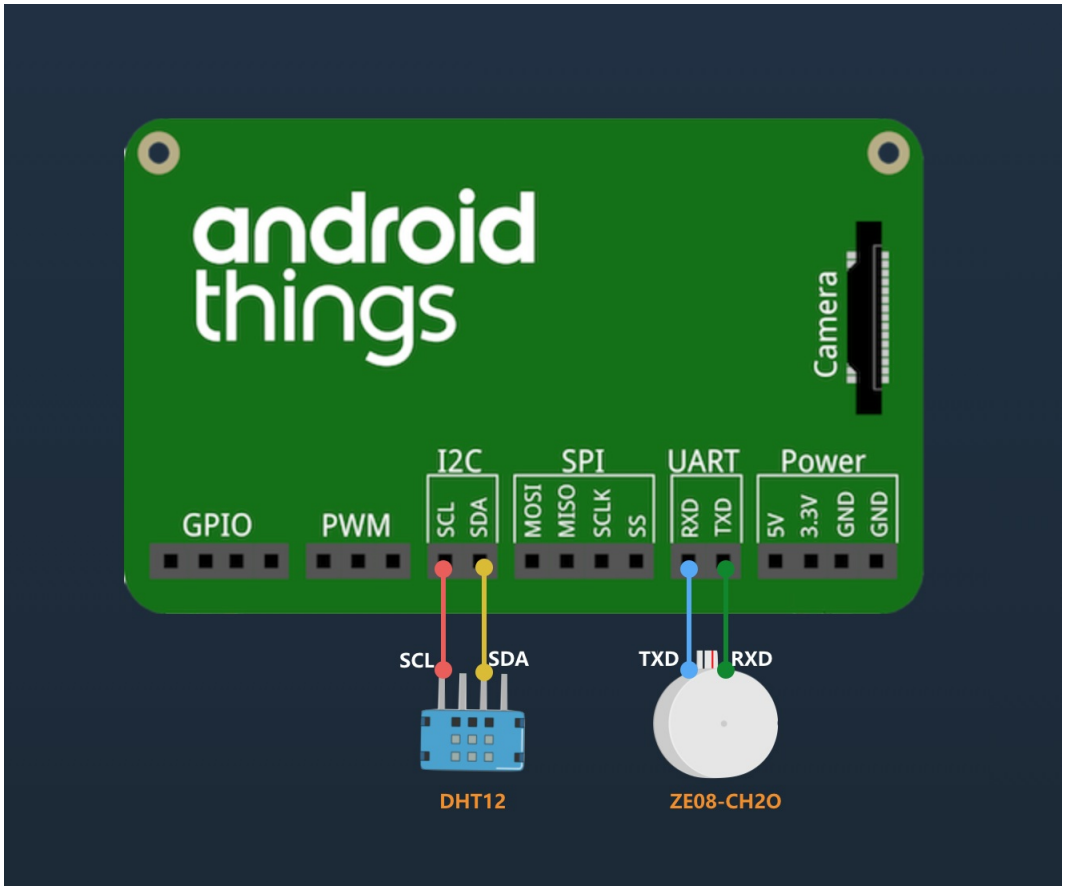
- Hardware list for the project

  The following table describes the hardware used in the indoor air test project.

Hardware	Picture	Remarks
NXP Pico i.MX7D Development board		Based on Android things 1.0. For more information, see the Developer Guide for NXP Pico i.MX7D. **Note** You can also use Raspberry Pi instead. For more information, see Remotely control a Raspberry Pi server.
DHT12 Temperature and humidity sensor		Supports the inter-integrated circuit (I2C) data communication method.

Hardware	Picture	Remarks
ZE08-CH2O Formaldehyde detection sensor		Supports the universal asynchronous receiver-transmitter (UART) data communication method.

- Hardware connection diagram



  - Connect the SCL (clock line) and SDA (data line) pins of the temperature and humidity sensor (DHT12) to the I2C SCL and SDA pins of the development board.

- Connect the TXD (transmit data) pin of the formaldehyde detection sensor (ZE08-CH2O) to the RXD (receive data) pin of the development board, and connect the RXD pin of ZE08-CH2O to the TXD pin of the development board.

Create a product and a device in the IoT Platform console

1. Log on to the IoT Platform console.
2. Create a product.
   i.
   ii. In the left-side navigation pane, choose **Devices** > **Products**.
   iii. On the **Products** page, click **Create Product** to create a product. For more information, see Create a product.
3. Create a TSL model.
   i. On the page that appears after the product is created, click **Create TSL**.
   ii. On the **Product Details** page, click the **Define Feature** tab. Click **Edit Draft** and then **Add Self-defined Feature**.
   iii. Add the properties described in the following table.

Property	Identifier	Data Type	Valid value	Description
Temperature	temperature	float	-50 to 100	The temperature data collected by the temperature and humidity sensor DHT12.
Humidity	humidity	float	0 to 100	The humidity data collected by the temperature and humidity sensor DHT12.
Formaldehyde concentration	ch2o	double	0 to 3	The formaldehyde concentration collected by the formaldehyde detection sensor ZE08-CH2O.

   iv. Click **Release online** to publish the TSL model.
4. Create a device.

   On the **Devices** page, click **Add Device** to add a device to the product. For more information, see Create a device.

Develop an Android Things device

1. Use Android Studio to create an Android Things project and add the network permission.

```
<uses-permission android:name="android.permission.INTERNET" />
```

2. Add `eclipse.paho.mqtt` to the Gradle file.

```
implementation 'org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.2.0'
```

3. Configure the device to read data from the temperature and humidity sensor DHT12 by using I2C.

```
private void readDataFromI2C() {
        try {
            byte[] data = new byte[5];
            i2cDevice.readRegBuffer(0x00, data, data.length);
            // check data
            if ((data[0] + data[1] + data[2] + data[3]) % 256 != data[4]) {
                humidity = temperature = 0;
                return;
            }
            // humidity data
            humidity = Double.valueOf(String.valueOf(data[0]) + "." + String.valueOf(da
ta[1]));
            Log.d(TAG, "humidity: " + humidity);
            // temperature data
            if (data[3] < 128) {
                temperature = Double.valueOf(String.valueOf(data[2]) + "." + String.val
ueOf(data[3]));
            } else {
                temperature = Double.valueOf("-" + String.valueOf(data[2]) + "." + Stri
ng.valueOf(data[3] - 128));
            }
            Log.d(TAG, "temperature: " + temperature);
        } catch (IOException e) {
            Log.e(TAG, "readDataFromI2C error " + e.getMessage(), e);
        }
    }
```

4. Configure the device to read data from the formaldehyde detection sensor ZE08-CH2O by using UART.

```
try {
                // data buffer
                byte[] buffer = new byte[9];
                while (uartDevice.read(buffer, buffer.length) > 0) {
                    if (checkSum(buffer)) {
                        ppbCh2o = buffer[4] * 256 + buffer[5];
                        ch2o = ppbCh2o / 66.64 * 0.08;
                    } else {
                        ch2o = ppbCh2o = 0;
                    }
                    Log.d(TAG, "ch2o: " + ch2o);
                }
            } catch (IOException e) {
                Log.e(TAG, "Ze08CH2O read data error " + e.getMessage(), e);
            }
```

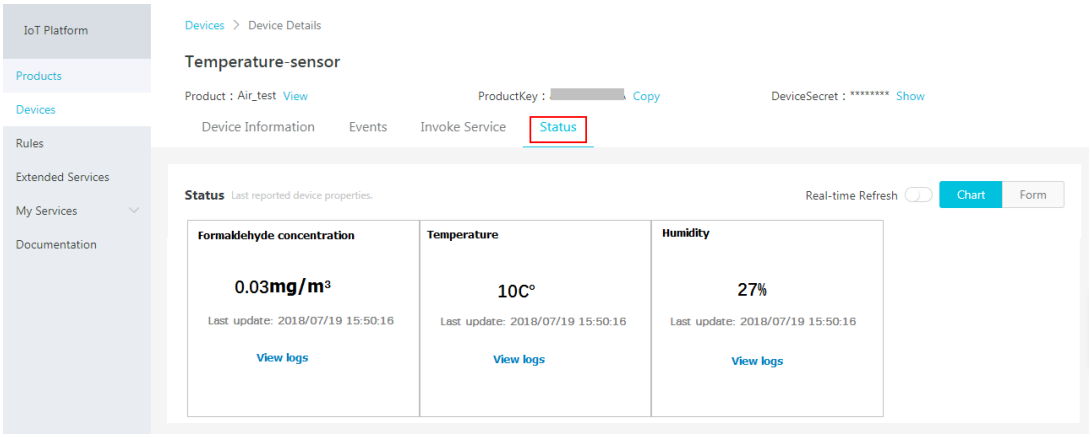
5. Connect the device to IoT Platform and report data.

```
/*
Payload format
{
  "id": 123243,
  "params": {
    "temperature": 25.6,
    "humidity": 60.3,
    "ch2o": 0.048
  },
  "method": "thing.event.property.post"
}
*/
MqttMessage message = new MqttMessage(payload.getBytes("utf-8"));
message.setQos(1);
String pubTopYourPc = "/sys/${YourProductKey}/${YourDeviceName}/thing/event/property/post";
mqttClient.publish(pubTopic, message);
```

You can visit aliyun-iot-androidthings-nxp on GitHub to download the complete sample project.

View real-time data of the device

After the device is started, log on to the IoT Platform console. On the **Device Details** page, view the real-time property data of the device in the **Status** section.



1.9. Connect an RTOS device to IoT Platform by using a TCP communication module

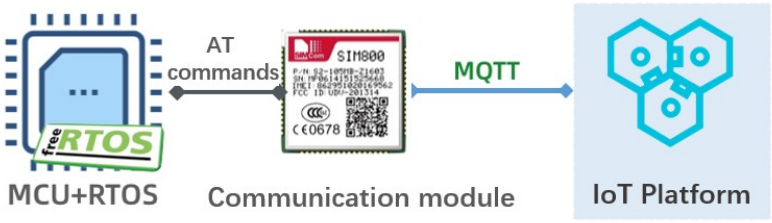
1.9.1. Overview

This topic describes how to use a device SDK for C-based application to connect a microcontroller (MCU) device on which a real-time operating system (RTOS) runs to IoT Platform.

Most electrical devices use one or more MCUs on which an RTOS runs. These electrical devices include industrial automation devices, data collection devices, real-time control devices, and home appliances. To connect these electrical devices to IoT Platform, use device SDK for C that is provided by IoT Platform.

Implementation

Connect an MCU to a communication module and use AT commands for communication between the MCU and the communication module. Use device SDK for C to connect the communication module to IoT Platform and enable communication between each other.



Prepare software and hardware

You must prepare the following items.

- A NUCLEO-F103RB demo board that is equipped with an STM32F103 MCU from ST.
- An SIM800C mini V2.0 demo board that is equipped with a SIM800C communication module from SIMcom.
- The IAR Embedded Workbench for ARM development environment.

Procedure

Create a product and add a device

Build a development environment for devices

Develop for a device

1.9.2. Create a product and add a device

This topic describes how to create a product, add a device to the product, and obtain certificate information about the device in the IoT Platform console. The certificate information includes the ProductKey, DeviceName, and DeviceSecret parameters. Certificate information about a device must be added to a device SDK-based project. If a device attempts to communicate with IoT Platform, IoT Platform uses the certificate information about the device to authenticate the device.

Procedure

1. Log on to the IoT Platform console.
2. Create a product.
    i. In the left-side navigation pane, choose **Devices > Products**.

ii. On the **Products** page, click **Create Product**, specify the required parameters, and click **Save**.

Parameter	Description
Product Name	The name of the product.
[DO NOT TRANSLATE]	[DO NOT TRANSLATE]
Node Type	Select **Directly Connected Device**.
Network Connection Method	Select **Wi-Fi**.
Data Type	Select **ICA Standard Data Format (Alink JSON)**.
Authentication Mode	Select **Device Secret**.

3. Add a device to the product.
   i. In the left-side navigation pane, choose **Devices > Devices**.
   ii. On the **Devices** page, click **Add Device** to add a device to the product.

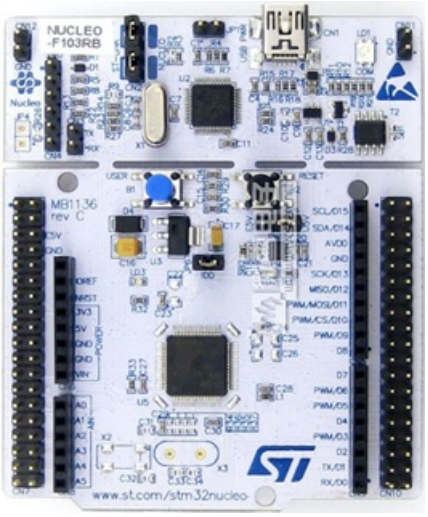
1.9.3. Build a development environment for devices

This topic describes how to connect a microcontroller (MCU) to a demo board that includes a communication module. It also describes how to build a development environment, create a project, import a device SDK, and configure the project.

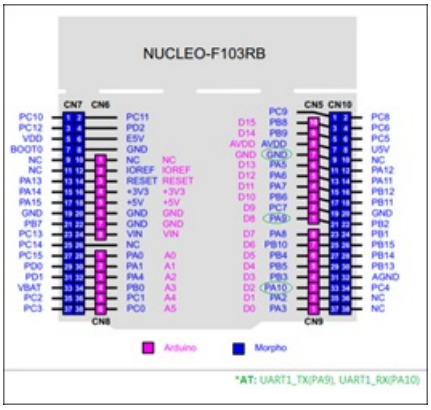
Context

The following figure shows the demo boards.

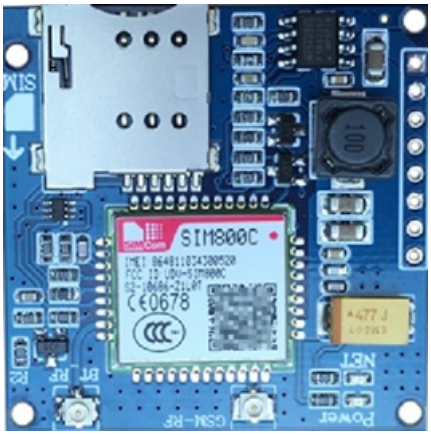
- NUCLEO-F103RB



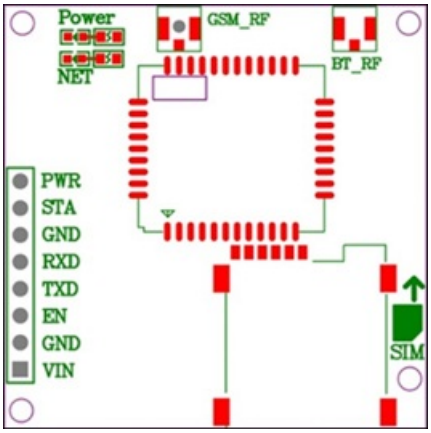
The following figure shows pins for the demo board.



- SIM800C mini v2.0



The following figure shows pins for the demo board.

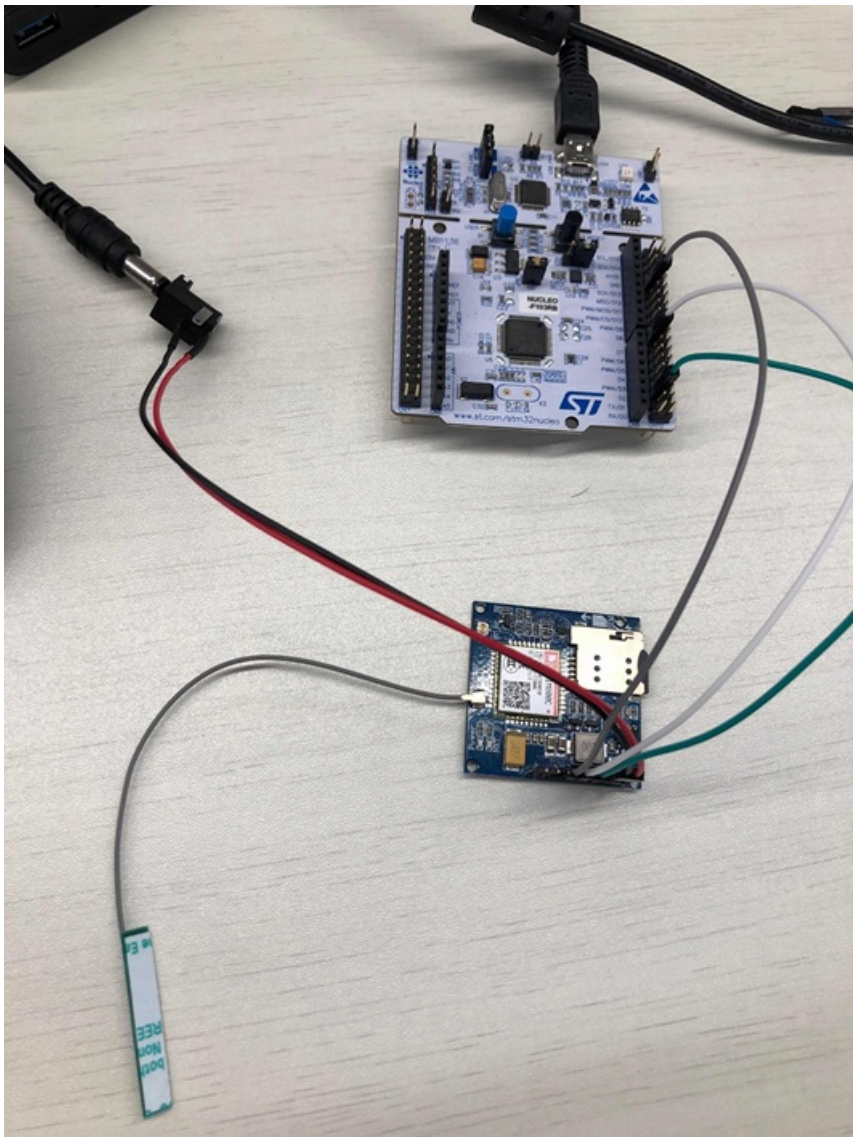


Pin	Description
PWR	Power switch. Automatic power-on is enabled by default.
STA	Status.
GND	Ground.
RXD	Receive data.
TXD	Transmit data.

Pin	Description
EN	Enable.
VIN	5 V to 18 V input.

Connect demo boards

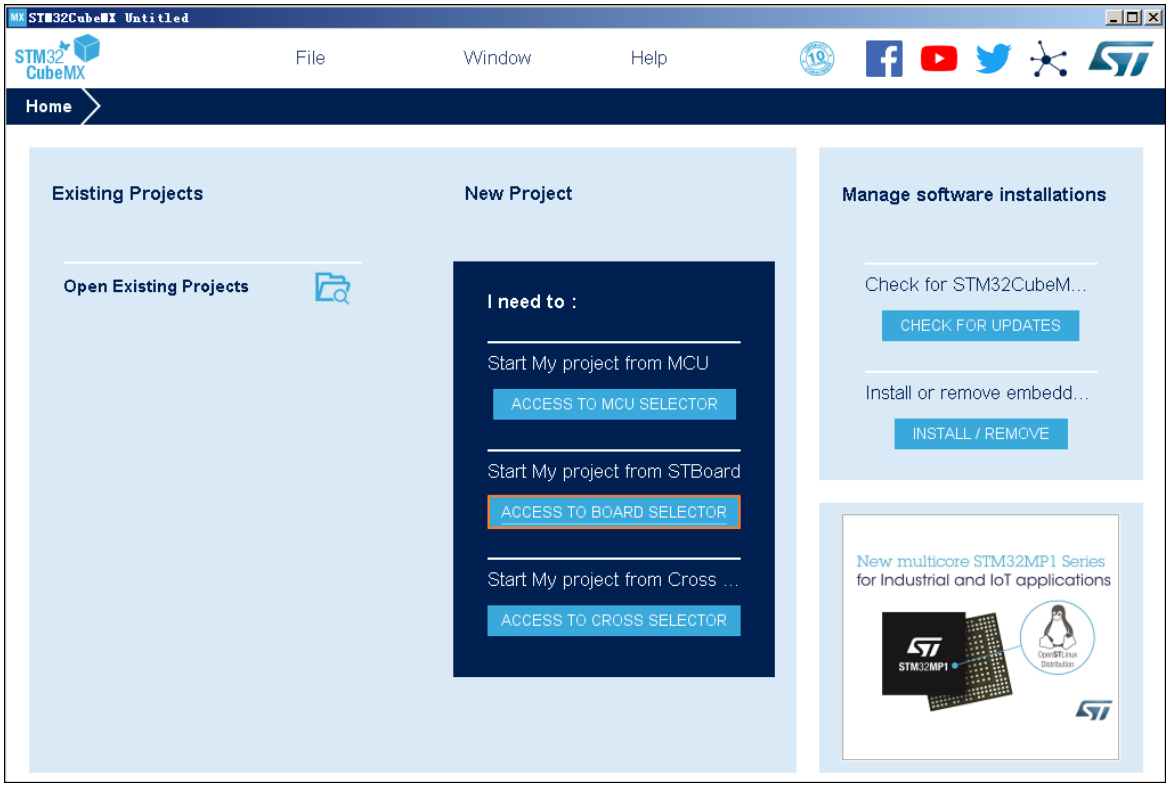
Connect the RXD pin of each demo board to the TXD pin of each demo board. This allows you to establish a connection and run AT commands. The following figure shows how to connect the demo boards.



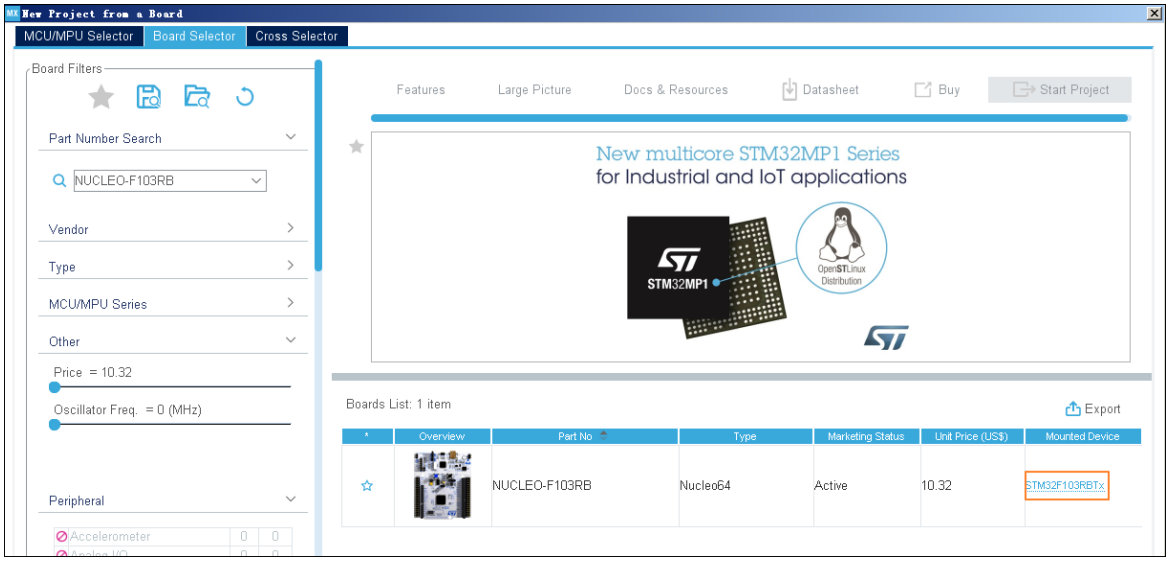
Build a development environment

This topic uses the STM32CubeMX tool as an example. For more information, see STM32Cube Ecosystem.

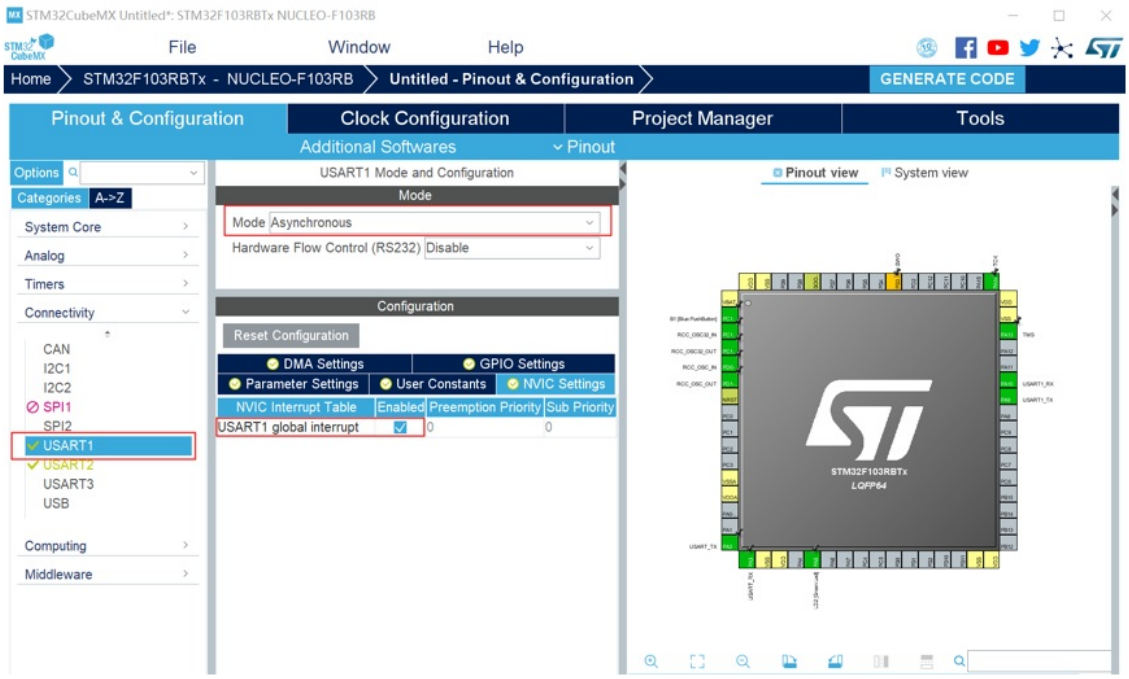
1. Open the STM32CubeMX tool and click New Project.



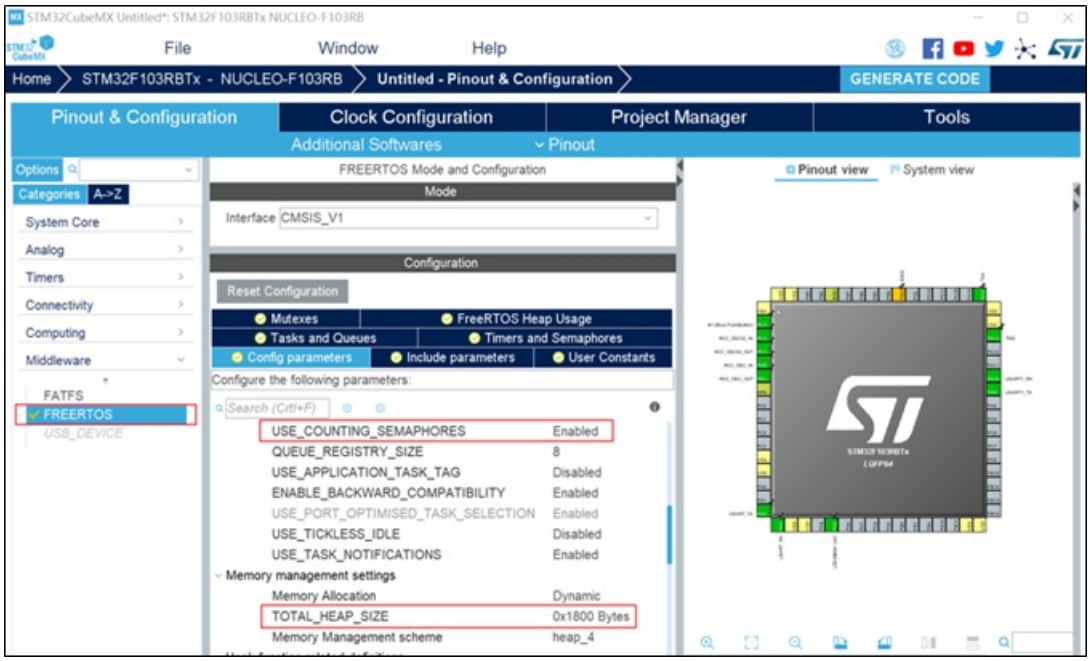
2. On the **Board Seletor** tab, search for **NUCLEO-F103RB** and click **STM32F103RBTx**.



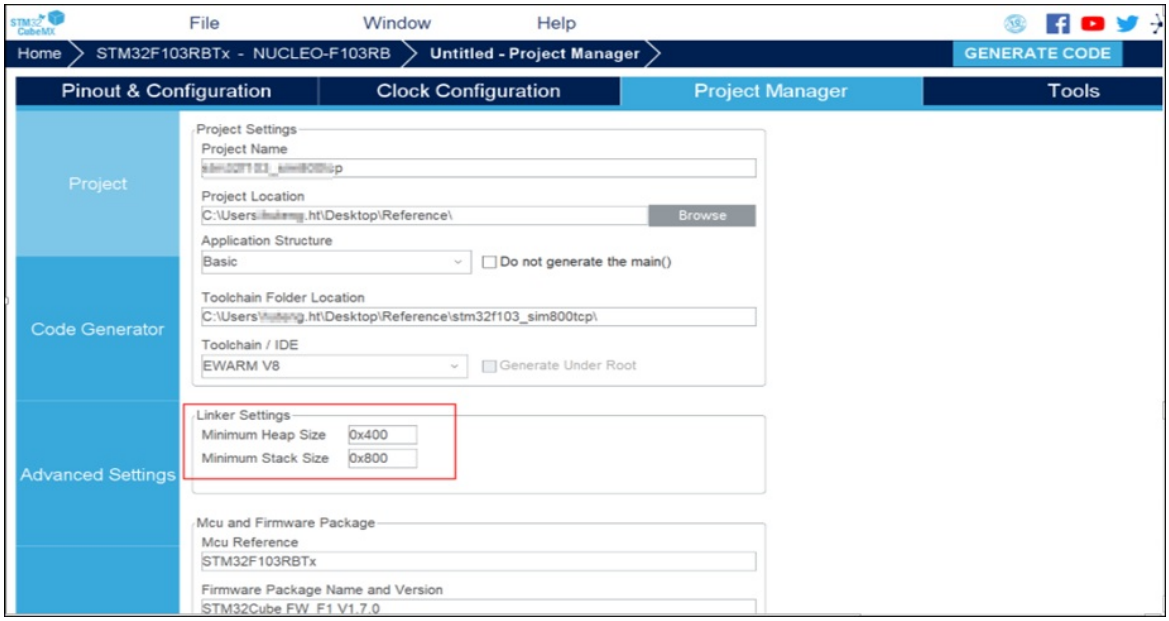
3. In the upper-right corner, click **Start Project**.
4. In the left-side navigation pane, click **Connectivity**, and select **USART1** as the port for communication between an MCU and a communication module. Proceed as follows.
   - Set **Mode** to **Asynchronous**.
   - In the **Configuration** section, set the following parameters.
     - On the **GPIO Settings** tab, select **PA9** and **PA10** in the Pin field.
     - On the **NVIC Settings** tab, set **USART1 global interrupt** to **Enabled**.



5. In the left-side navigation pane, click **Middleware**, select **FREERTOS**, set USE_COUNTING_SEMAPHORES to Enabled, and set TOTAL_HEAP_SIZE. Each thread can share the heap.



6. On the **Project Manager** tab, complete the settings in the **Project** section.



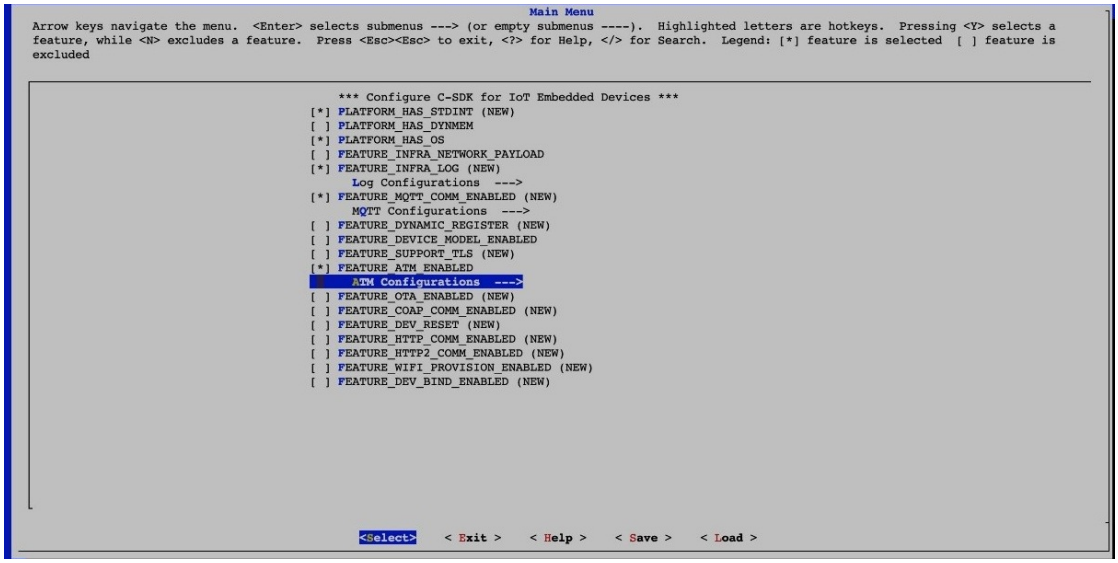
7. In the upper-right corner of the page, click **GENERATE CODE** to generate code for the project.

1.9.4. Develop for a device

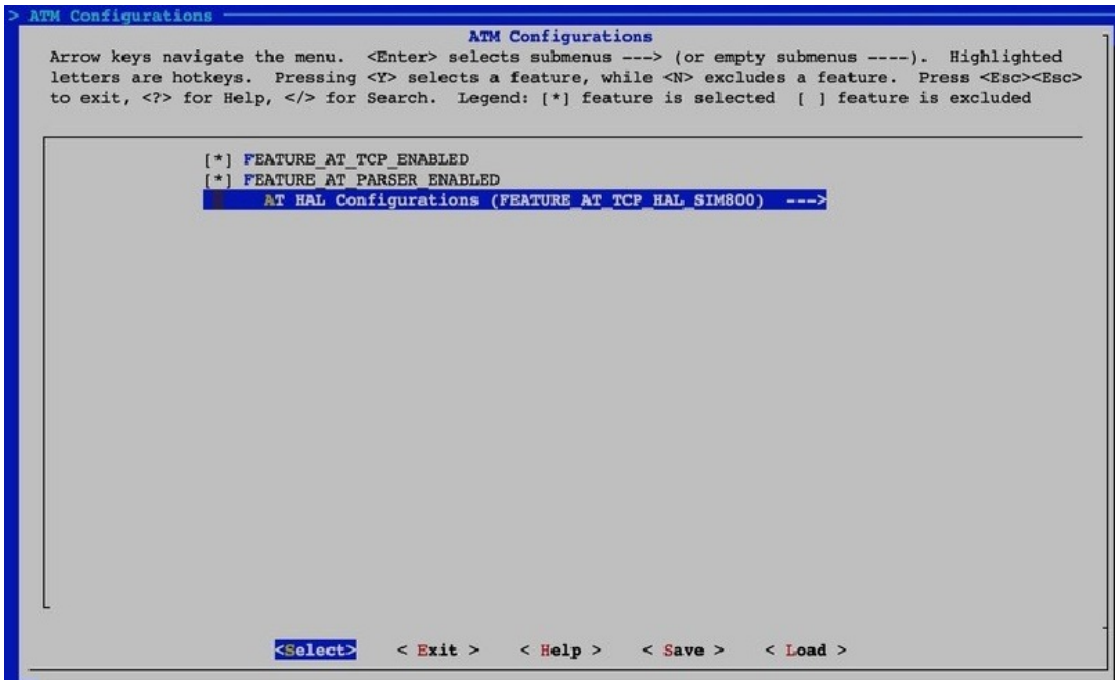
Use Link SDK for C to connect a communication module to IoT Platform.

Configure a device SDK

1. Download Link SDK 3.0.1 for C.
2. Extract code from the SDK package. This topic uses Linux as an example.
   i. Run the **make menuconfig** command.
   ii. Select **ATM Configurations** and click **Select**.



iii. Select **AT HAL Configurations** and click **Select**.



iv. Set the following parameters:

```
FEATURE_PLATFORM_HAS_STDINT=y
FEATURE_PLATFORM_HAS_OS=y
FEATURE_INFRA_STRING=y
FEATURE_INFRA_NET=y
FEATURE_INFRA_LIST=y
FEATURE_INFRA_LOG=y
FEATURE_INFRA_LOG_ALL_MUTED=y
FEATURE_INFRA_LOG_MUTE_FLW=y
FEATURE_INFRA_LOG_MUTE_DBG=y
FEATURE_INFRA_LOG_MUTE_INF=y
FEATURE_INFRA_LOG_MUTE_WRN=y
FEATURE_INFRA_LOG_MUTE_ERR=y
FEATURE_INFRA_LOG_MUTE_CRT=y
FEATURE_INFRA_TIMER=y
FEATURE_INFRA_SHA256=y
FEATURE_INFRA_REPORT=y
FEATURE_INFRA_COMPAT=y
FEATURE_DEV_SIGN=y
FEATURE_MQTT_COMM_ENABLED = y
FEATURE_MQTT_DEFAULT_IMPL=y
FEATURE_MQTT_DIRECT = y
FEATURE_DEVICE_MODEL_CLASSIC=y
FEATURE_ATM_ENABLED=y
FEATURE_AT_TCP_ENABLED=y
FEATURE_AT_PARSER_ENABLED=y
FEATURE_AT_TCP_HAL_SIM800=y
```

v. After the configuration is complete, run the **./extract.sh** script to extract code.

```
[redacted]:~/c-sdk$ ./extract.sh
. Download request sent, waiting respond ...
. Respond generating, wait longer
. Retried 1/20

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  107k  100  107k    0     0   408k      0 --:--:-- --:--:-- --:--:--  408k

Please pick up extracted source files in [/disk1/huteng/c-sdk/output]
```

The code is saved in the *output/eng* directory.

```
[redacted]:~/c-sdk/output/eng$ ls
atm  dev_sign  infra  mqtt  sdk_include.h  wrappers
```

The following table lists subdirectories that reside in the output/eng directory.

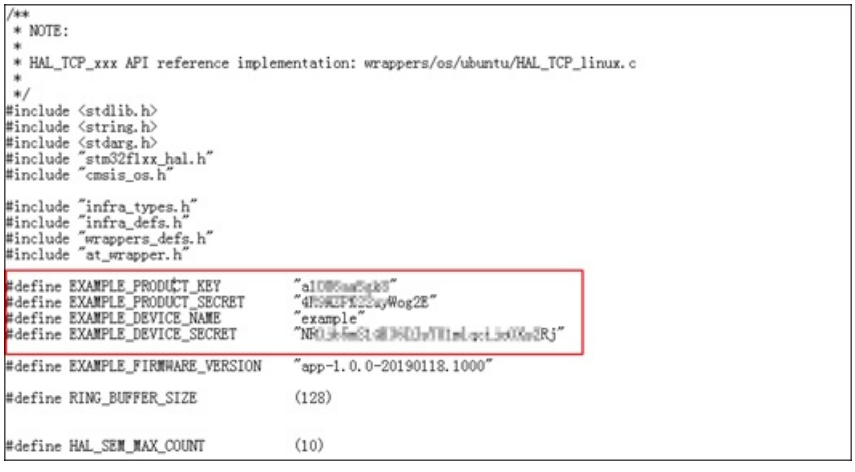
Directory	Description
atm	Includes code that you can use to send and receive AT commands.
dev_sign	Includes code that you can use to authenticate devices.
infra	Includes code that you can use to implement internal functions.
mqtt	Includes code that you can use to implement the Message Queuing Telemetry Transport (MQTT) protocol.
wrappers	Includes code that you can use to implement hardware abstraction layer (HAL) functions.

vi. In the *wrappers* directory, create the *wrappers.c* file. Implement the following HAL functions in the file.

```
int32_t HAL_AT_Uart_Deinit(uart_dev_t *uart)
int32_t HAL_AT_Uart_Init(uart_dev_t *uart)
int32_t HAL_AT_Uart_Recv(uart_dev_t *uart, void *data, uint32_t expect_size, uint32
_t *recv_size, uint32_t timeout)
int32_t HAL_AT_Uart_Send(uart_dev_t *uart, const void *data, uint32_t size, uint32_
t timeout)
int HAL_GetDeviceName(char device_name[IOTX_DEVICE_NAME_LEN])
int HAL_GetDeviceSecret(char device_secret[IOTX_DEVICE_SECRET_LEN])
int HAL_GetFirmwareVersion(char *version)
int HAL_GetProductKey(char product_key[IOTX_PRODUCT_KEY_LEN])
void *HAL_Malloc(uint32_t size);
void HAL_Free(void *ptr);
void *HAL_MutexCreate(void)
void ls_osa_mutex_destroy(void *mutex)
void HAL_MutexLock(void *mutex)
void HAL_MutexUnlock(void *mutex)
void *HAL_SemaphoreCreate(void)
void HAL_SemaphoreDestroy(void *sem)
void HAL_SemaphorePost(void *sem)
int HAL_SemaphoreWait(void *sem, uint32_t timeout_ms)
int HAL_ThreadCreate(void **thread_handle,
                     void *(*work_routine)(void *),
                     void *arg,
                     hal_os_thread_param_t *hal_os_thread_param,
                     int *stack_used)
void HAL_SleepMs(uint32_t ms)
void ls_osa_print(const char *fmt, ...)
int HAL_Snprintf(char *str, const int len, const char *fmt, ...)
uint64_t HAL_UptimeMs(void)
```

Download a sample wrappers.c file.

The following figure shows how to specify the certificate information about the serial device in the file.

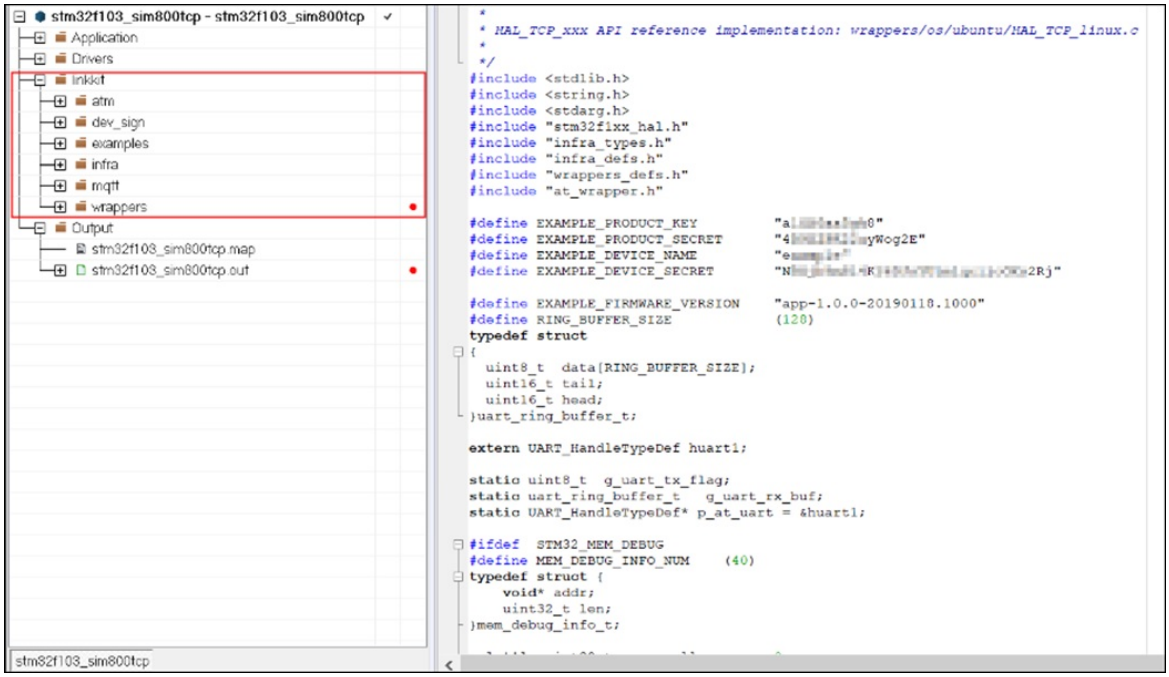


```
/**
 * NOTE:
 *
 * HAL_TCP_xxx API reference implementation: wrappers/os/ubuntu/HAL_TCP_linux.c
 *
 */
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include "stm32f1xx_hal.h"
#include "cmsis_os.h"

#include "infra_types.h"
#include "infra_defs.h"
#include "wrappers_defs.h"
#include "at_wrapper.h"

#define EXAMPLE_PRODUCT_KEY         "a1[redacted]"
#define EXAMPLE_PRODUCT_SECRET      "4[redacted]xWog2E"
#define EXAMPLE_DEVICE_NAME         "example"
#define EXAMPLE_DEVICE_SECRET       "NR[redacted]Rj"

#define EXAMPLE_FIRMWARE_VERSION    "app-1.0.0-20190118.1000"

#define RING_BUFFER_SIZE            (128)


#define HAL_SEM_MAX_COUNT           (10)
```

Note If you are not using the NUCLEO-F103RB demo board, set `FEATURE_AT_TCP_HAL_SIM800=n`. You also need to implement the following HAL functions in the *wrappers.c* file.

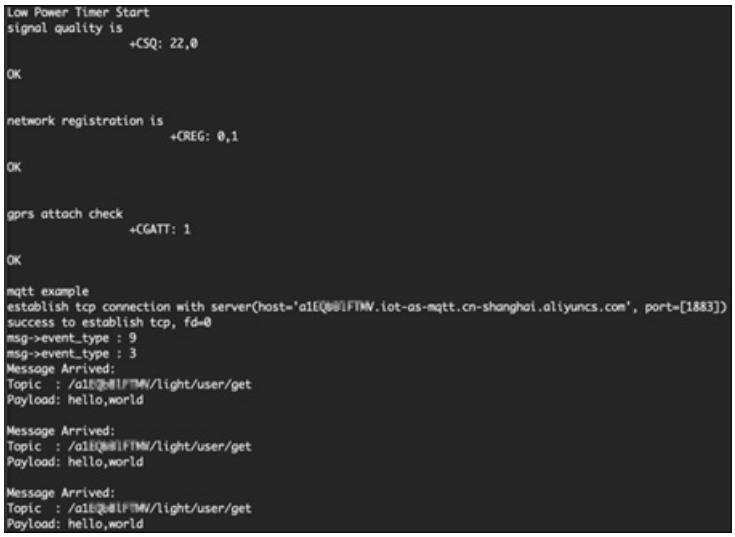
```
int HAL_AT_CONN_Close(int fd, int32_t remote_port)
int HAL_AT_CONN_Deinit(void)
int HAL_AT_CONN_DomainToIp(char *domain, char ip[16])
int HAL_AT_CONN_Init(void)
int HAL_AT_CONN_Send(int fd, uint8_t *data, uint32_t len, char remote_ip[16], i
nt32_t remote_port, int32_t timeout)
int HAL_AT_CONN_Start(at_conn_t *conn)
int32_t HAL_AT_Uart_Deinit(uart_dev_t *uart)
int32_t HAL_AT_Uart_Init(uart_dev_t *uart)
int32_t HAL_AT_Uart_Recv(uart_dev_t *uart, void *data, uint32_t expect_size, ui
nt32_t *recv_size, uint32_t timeout)
int32_t HAL_AT_Uart_Send(uart_dev_t *uart, const void *data, uint32_t size, uin
t32_t timeout)
int HAL_GetDeviceName(char device_name[IOTX_DEVICE_NAME_LEN])
int HAL_GetDeviceSecret(char device_secret[IOTX_DEVICE_SECRET_LEN])
int HAL_GetFirmwareVersion(char *version)
int HAL_GetProductKey(char product_key[IOTX_PRODUCT_KEY_LEN])
void *HAL_Malloc(uint32_t size);
void HAL_Free(void *ptr);
void *HAL_MutexCreate(void)
void ls_osa_mutex_destroy(void *mutex)
void HAL_MutexLock(void *mutex)
void HAL_MutexUnlock(void *mutex)
void *HAL_SemaphoreCreate(void)
void HAL_SemaphoreDestroy(void *sem)
void HAL_SemaphorePost(void *sem)
int HAL_SemaphoreWait(void *sem, uint32_t timeout_ms)
int HAL_ThreadCreate(void **thread_handle,
                     void *(*work_routine)(void *),
                     void *arg,
                     hal_os_thread_param_t *hal_os_thread_param,
                     int *stack_used)
void HAL_SleepMs(uint32_t ms)
void ls_osa_print(const char *fmt, ...)
int HAL_Snprintf(char *str, const int len, const char *fmt, ...)
uint64_t HAL_UptimeMs(void)
```

3. Add the file to the IAR project.



4. Run the project to perform a test.

   After the project is completed, the following figure shows device logs.



   In the IoT Platform console, choose **Maintenance > Device Log**. The Device Log page also shows logs that are generated each time the device reports data to IoT Platform.

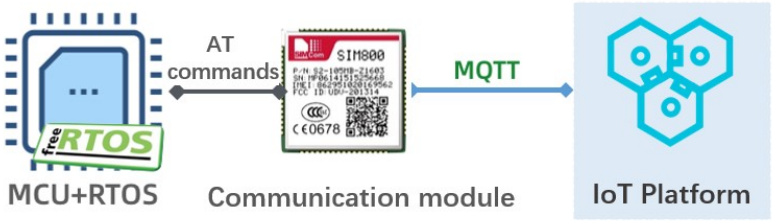
1.10. Connect a bare-metal device to IoT Platform by using TCP

1.10.1. Overview

This topic describes how to use device SDK for C to connect a bare-metal microcontroller device to IoT Platform by using Message Queuing Telemetry Transport (MQTT).

Implementation

Connect an MCU to a communication module and use AT commands for communication between the MCU and the communication module. Use device SDK for C to connect the communication module to IoT Platform and enable communication between each other.



Prepare software and hardware

You must prepare the following items.

- A NUCLEO-F103RB demo board that is equipped with an STM32F103 MCU from ST.
- An SIM800C mini V2.0 demo board that is equipped with a SIM800C communication module from SIMcom.
- The IAR Embedded Workbench for ARM development environment.

Procedure

Create a product and add a device

Build a development environment for devices

Provision a device

1.10.2. Create a product and add a device

This topic describes how to create a product, add a device to the product, and obtain certificate information about the device in the IoT Platform console. The certificate information includes the ProductKey, DeviceName, and DeviceSecret parameters. Certificate information about a device must be added to a device SDK-based project. If a device attempts to communicate with IoT Platform, IoT Platform uses the certificate information about the device to authenticate the device.

Procedure

1. Log on to the IoT Platform console.
2. Create a product.
   i. In the left-side navigation pane, choose **Devices > Products**.

ii. On the **Products** page, click **Create Product**, specify the required parameters, and click **Save**.

Parameter	Description
Product Name	The name of the product.
[DO NOT TRANSLATE]	[DO NOT TRANSLATE]
Node Type	Select **Directly Connected Device**.
Network Connection Method	Select **Wi-Fi**.
Data Type	Select **ICA Standard Data Format (Alink JSON)**.
Authentication Mode	Select **Device Secret**.

3. Add a device to the product.
   i. In the left-side navigation pane, choose **Devices > Devices**.
   ii. On the **Devices** page, click **Add Device** to add a device to the product.

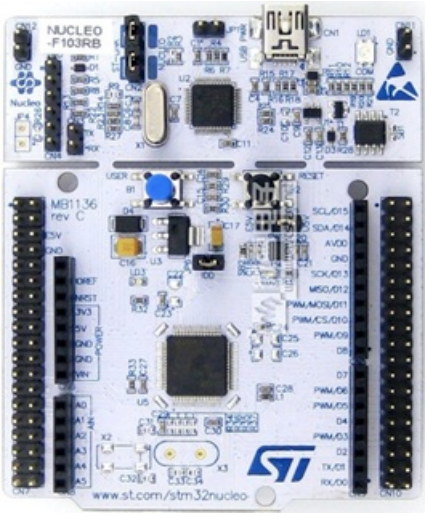
1.10.3. Build a development environment for devices

This topic describes how to connect a microcontroller (MCU) to a demo board that includes a communication module. It also describes how to build a development environment, create a project, import a device SDK, and configure the project.

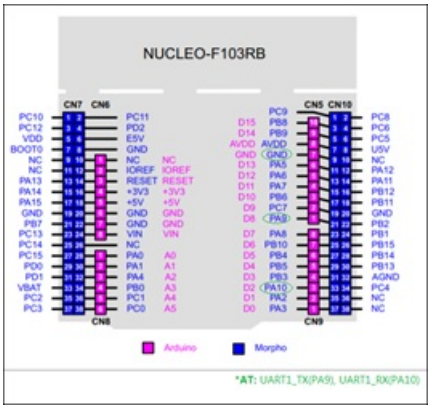
Context

The following figure shows the demo boards.

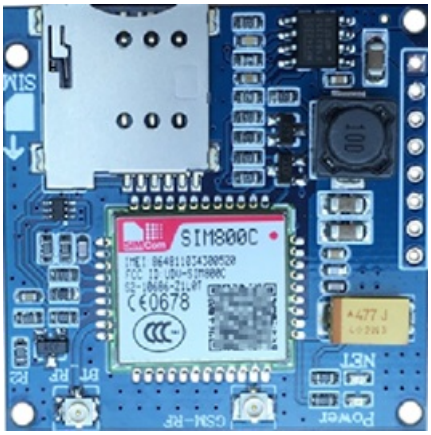
- NUCLEO-F103RB



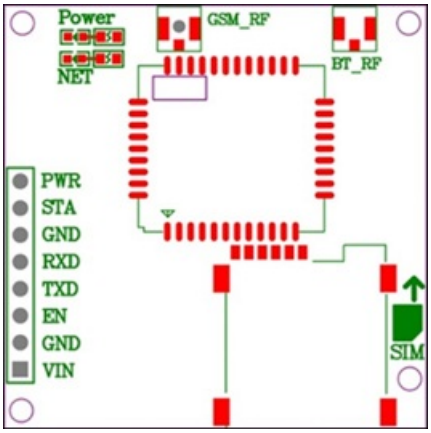
The following figure shows pins for the demo board.



- SIM800C mini v2.0



The following figure shows pins for the demo board.

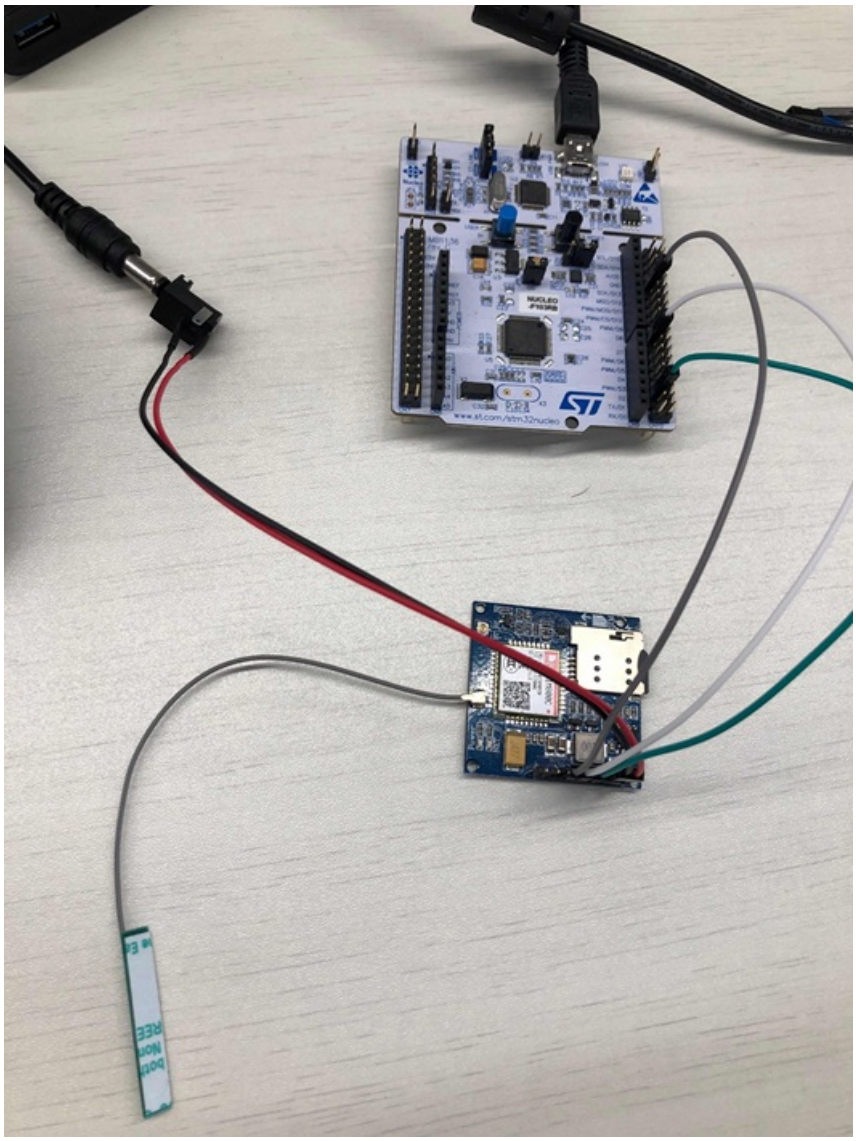


Pin	Description
PWR	Power switch. Automatic power-on is enabled by default.
STA	Status.
GND	Ground.
RXD	Receive data.
TXD	Transmit data.

Pin	Description
EN	Enable.
VIN	5 V to 18 V input.

Connect demo boards

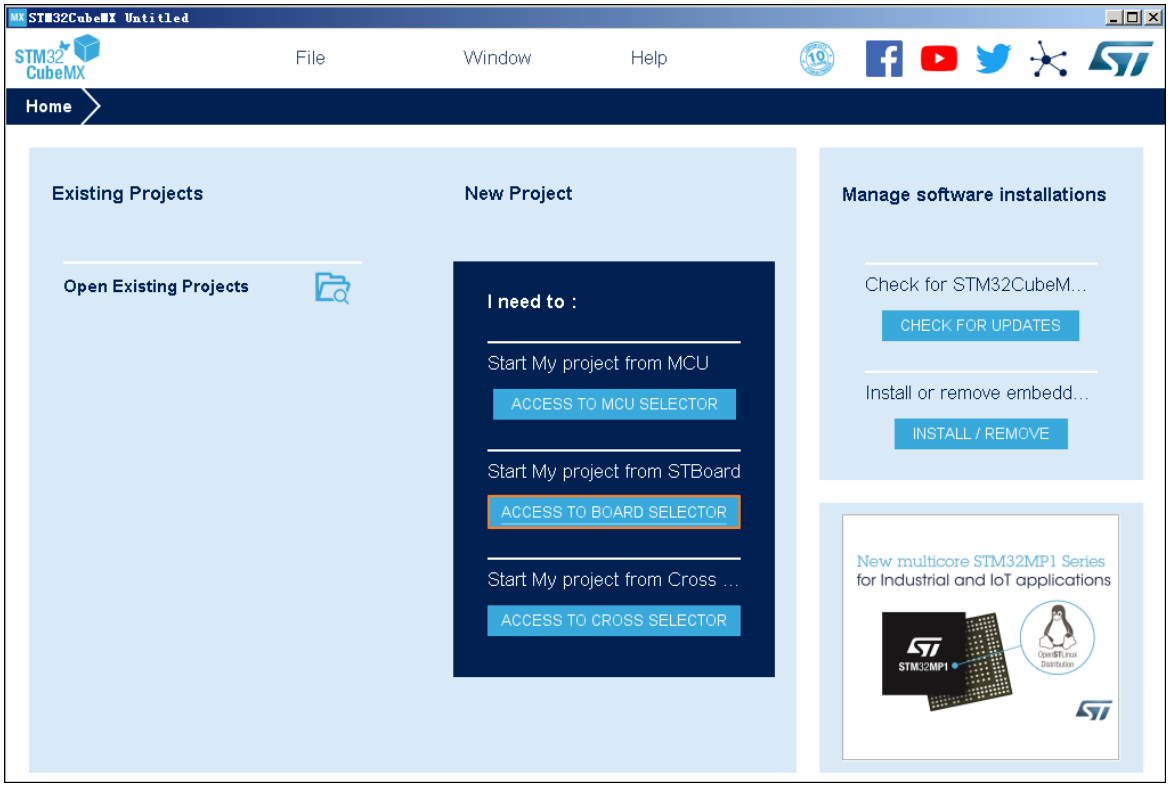
Connect the RXD pin of a demo board to the TXD pin of the other demo board. This allows you to establish a connection and run AT commands. The following figure shows how to connect the demo boards.



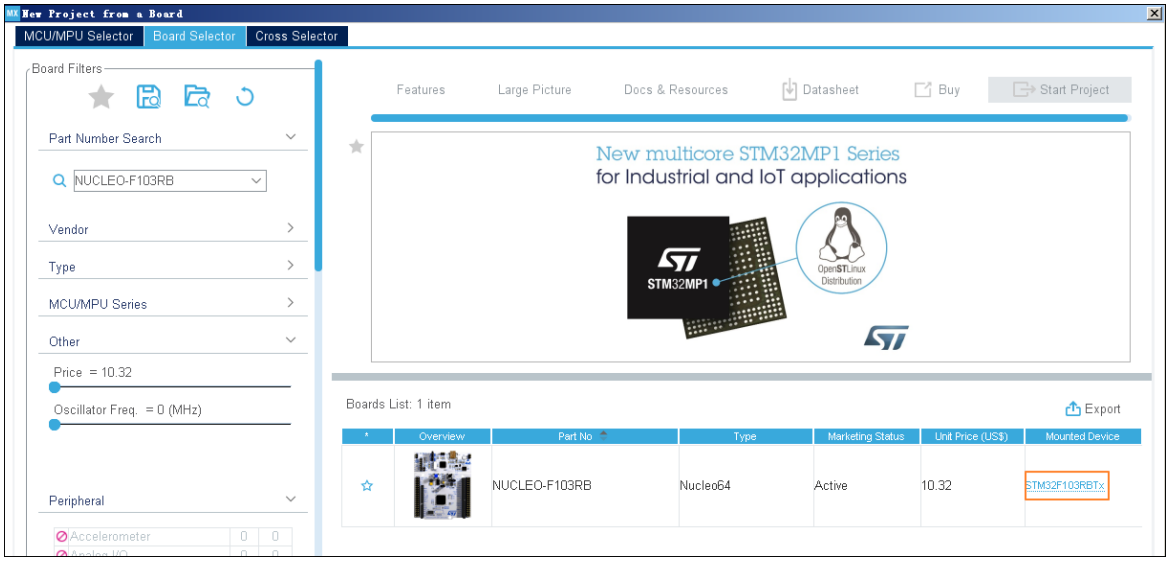
Build a development environment

This section uses the STM32CubeMX tool as an example. For more information, visit STM32Cube Ecosystem.

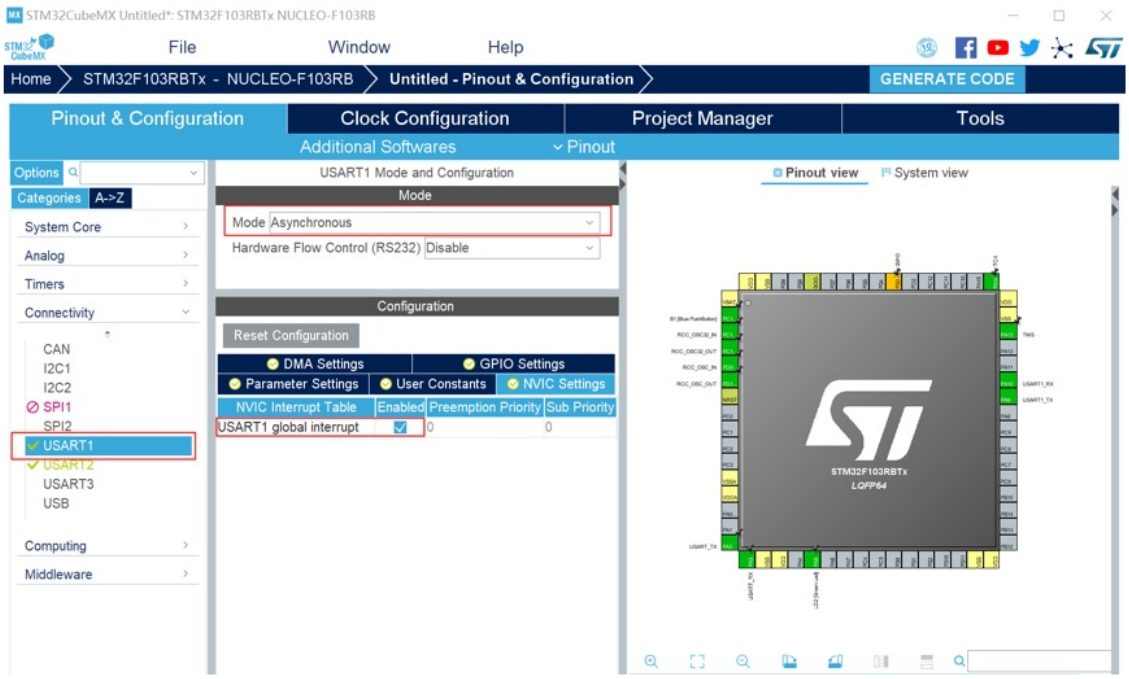
1. Open the STM32CubeMX tool and click New Project.



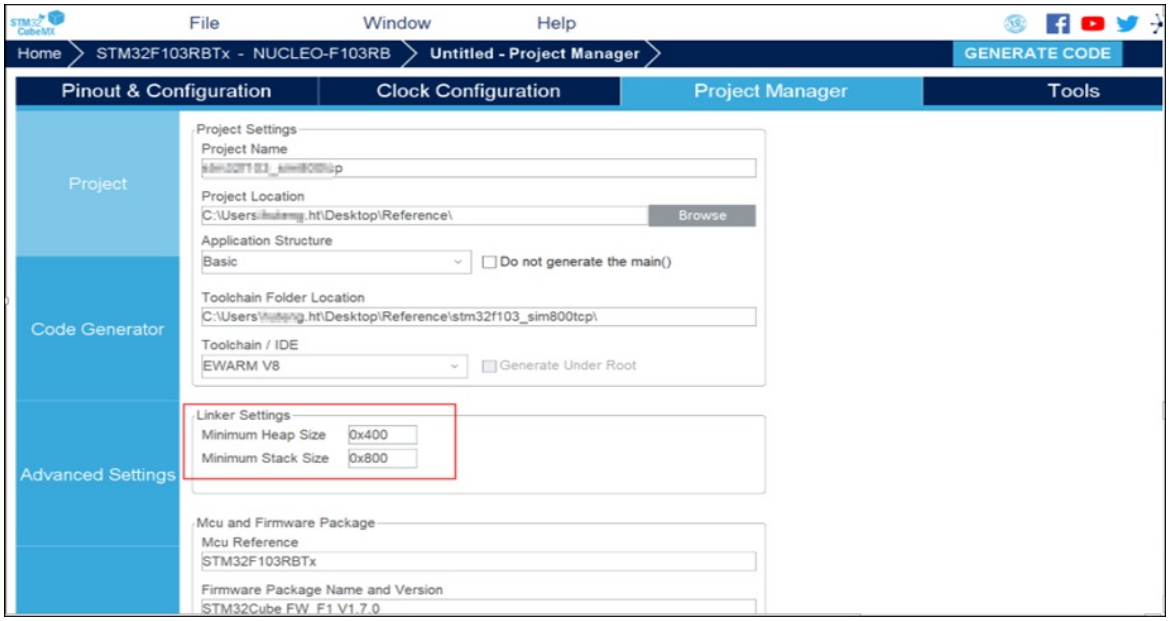
2. On the **Board Selector** tab, search for **NUCLEO-F103RB** and click **STM32F103RBTx**.



3. In the upper-right corner, click **Start Project**.
4. In the left-side navigation pane, click **Connectivity**, select **USART1** as the port for communication between the MCU and the communication module. Set the following parameters:
   - Set **Mode** to **Asynchronous**.
   - In the **Configuration** section, set the following parameters:
     - On the **GPIO Settings** tab, select **PA9** and **PA10** in the Pin field.
     - On the **NVIC Settings** tab, set **USART1 global interrupt** to **Enabled**.



5. On the **Project Manager** tab, specify the required parameters in the **Project** section.
   - Set **Toolchain/IDE** to **EWARM V8**.
   - Set **Heap/Stack size** based on your business requirements.



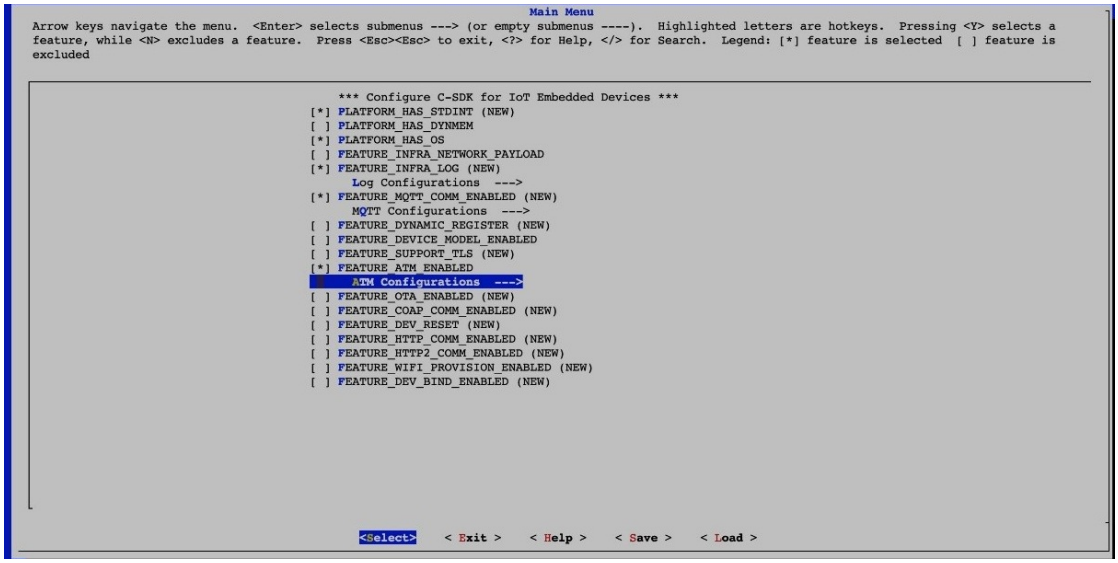
6. In the upper-right corner, click **GENERATE CODE** to generate code for the project.

1.10.4. Provision a device

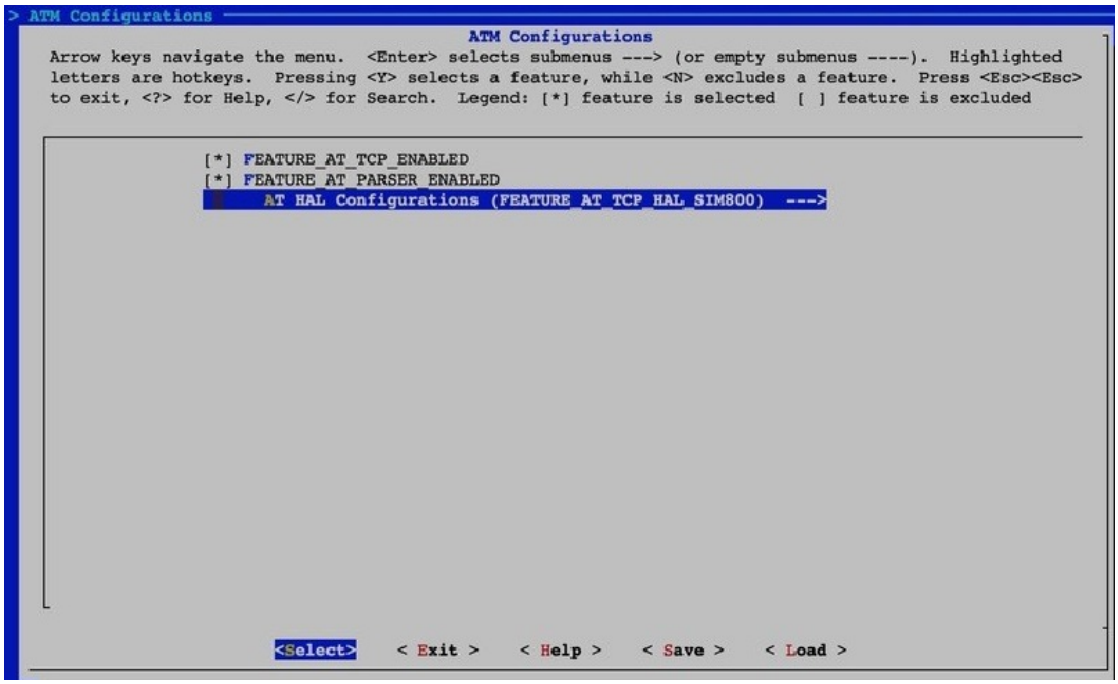
This section describes how to use the Link SDK for C to connect a communication module to IoT Platform.

Procedure

1. Download Link SDK 3.0.1 for C.
2. Extract code from the SDK package. This section uses Linux as an example.
   i. Run the **make menuconfig** command.
   ii. Select **ATM Configurations** and click **Select**.



   iii. Select **AT HAL Configurations** and click **Select**.



iv. Set the following parameters:

```
FEATURE_PLATFORM_HAS_STDINT=y
FEATURE_INFRA_STRING=y
FEATURE_INFRA_NET=y
FEATURE_INFRA_LIST=y
FEATURE_INFRA_LOG=y
FEATURE_INFRA_LOG_ALL_MUTED=y
FEATURE_INFRA_LOG_MUTE_FLW=y
FEATURE_INFRA_LOG_MUTE_DBG=y
FEATURE_INFRA_LOG_MUTE_INF=y
FEATURE_INFRA_LOG_MUTE_WRN=y
FEATURE_INFRA_LOG_MUTE_ERR=y
FEATURE_INFRA_LOG_MUTE_CRT=y
FEATURE_INFRA_TIMER=y
FEATURE_INFRA_SHA256=y
FEATURE_INFRA_REPORT=y
FEATURE_INFRA_COMPAT=y
FEATURE_DEV_SIGN=y
FEATURE_MQTT_COMM_ENABLED = y
FEATURE_MQTT_DEFAULT_IMPL=y
FEATURE_MQTT_DIRECT = y
FEATURE_DEVICE_MODEL_CLASSIC=y
FEATURE_ATM_ENABLED=y
FEATURE_AT_TCP_ENABLED=y
FEATURE_AT_PARSER_ENABLED=y
FEATURE_AT_TCP_HAL_SIM800=y
```

v. After the configuration is complete, run the **./extract.sh** script to exact the code.

```
[redacted]:~/c-sdk$ ./extract.sh
. Download request sent, waiting respond ...
. Respond generating, wait longer
. Retried 1/20

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  107k  100  107k    0     0   408k      0 --:--:-- --:--:-- --:--:--  408k

Please pick up extracted source files in [/disk1/huteng/c-sdk/output]
```

The code is saved in the *output/eng* directory.

```
[redacted]:~/c-sdk/output/eng$ ls
atm  dev_sign  infra  mqtt  sdk_include.h  wrappers
```

The following table lists the subdirectories that reside in the directory.

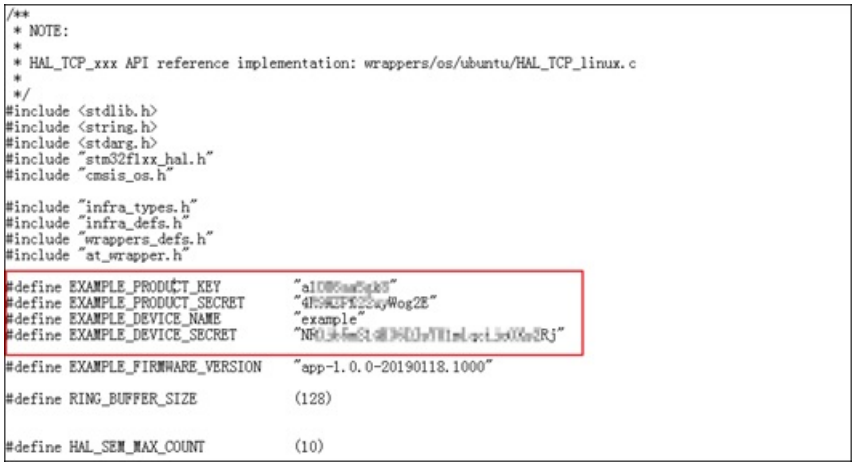
Directory	Description
atm	Includes code that you can use to send and receive AT instructions.
dev_sign	Includes code that you can use to authenticate devices.
infra	Includes code that you can use to implement internal functions.
mqtt	Includes code that you can use to implement the MQ Telemetry Transport (MQTT) protocol.
wrappers	Includes code that you can use to implement hardware abstraction layer (HAL) functions.

3. In the *wrappers* directory, create a file named *wrappers.c*. Implement the following HAL functions in the file.

```
int32_t HAL_AT_Uart_Deinit(uart_dev_t *uart)
int32_t HAL_AT_Uart_Init(uart_dev_t *uart)
int32_t HAL_AT_Uart_Recv(uart_dev_t *uart, void *data, uint32_t expect_size,
uint32_t *recv_size, uint32_t timeout)
int32_t HAL_AT_Uart_Send(uart_dev_t *uart, const void *data, uint32_t size,
uint32_t timeout)
int HAL_GetFirmwareVersion(char *version)
int HAL_GetDeviceName(char device_name[IOTX_DEVICE_NAME_LEN])
int HAL_GetDeviceSecret(char device_secret[IOTX_DEVICE_SECRET_LEN])
int HAL_GetProductKey(char product_key[IOTX_PRODUCT_KEY_LEN])
void *HAL_Malloc(uint32_t size);
void HAL_Free(void *ptr);
void *HAL_MutexCreate(void)
void HAL_MutexDestroy(void *mutex)
void HAL_MutexLock(void *mutex)
void HAL_MutexUnlock(void *mutex)
void HAL_Printf(const char *fmt, ...)
void HAL_SleepMs(uint32_t ms)
int HAL_Snprintf(char *str, const int len, const char *fmt, ...)
uint64_t HAL_UptimeMs(void)
```

Download a sample wrappers.c file.

The following figure shows how to specify the certificate information about the serial device in the file.

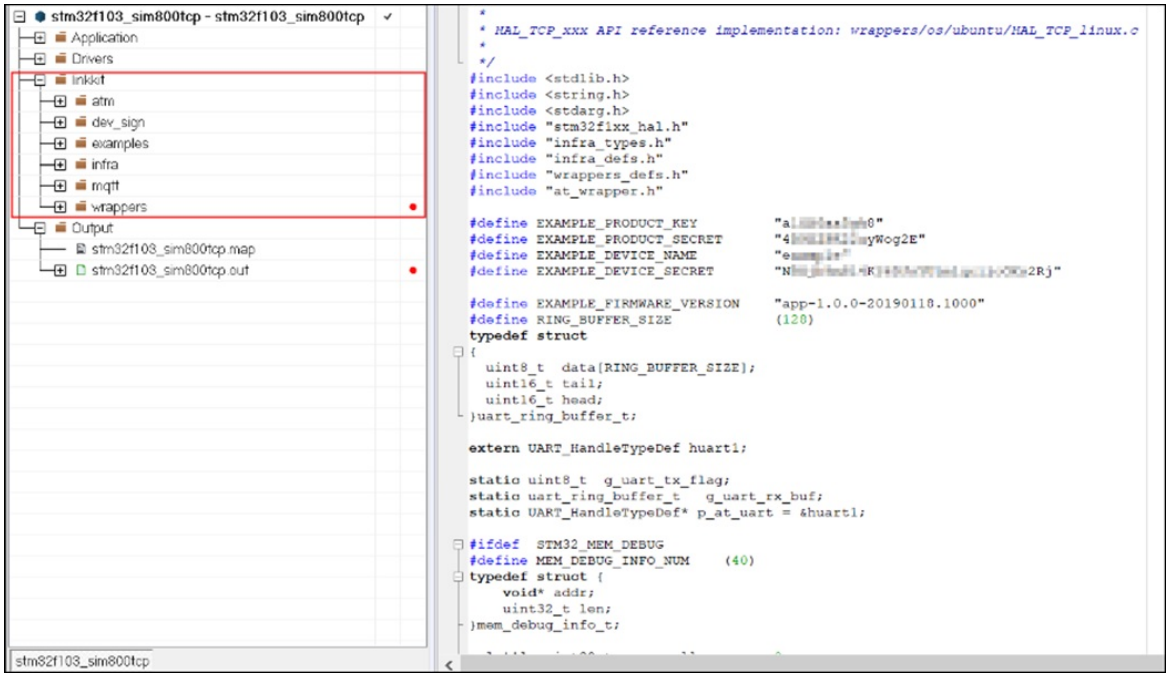


```
/**
 * NOTE:
 *
 * HAL_TCP_xxx API reference implementation: wrappers/os/ubuntu/HAL_TCP_linux.c
 *
 */
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include "stm32f1xx_hal.h"
#include "cmsis_os.h"

#include "infra_types.h"
#include "infra_defs.h"
#include "wrappers_defs.h"
#include "at_wrapper.h"

#define EXAMPLE_PRODUCT_KEY         "a1[illegible]"
#define EXAMPLE_PRODUCT_SECRET      "4[illegible]Wog2E"
#define EXAMPLE_DEVICE_NAME         "example"
#define EXAMPLE_DEVICE_SECRET       "NR[illegible]Rj"

#define EXAMPLE_FIRMWARE_VERSION    "app-1.0.0-20190118.1000"

#define RING_BUFFER_SIZE            (128)


#define HAL_SEM_MAX_COUNT           (10)
```

Note If you are not using the SIM800C communication module, add the `FEATURE_AT_TCP_HAL_SIM800=n` statement, and implement the following HAL functions.

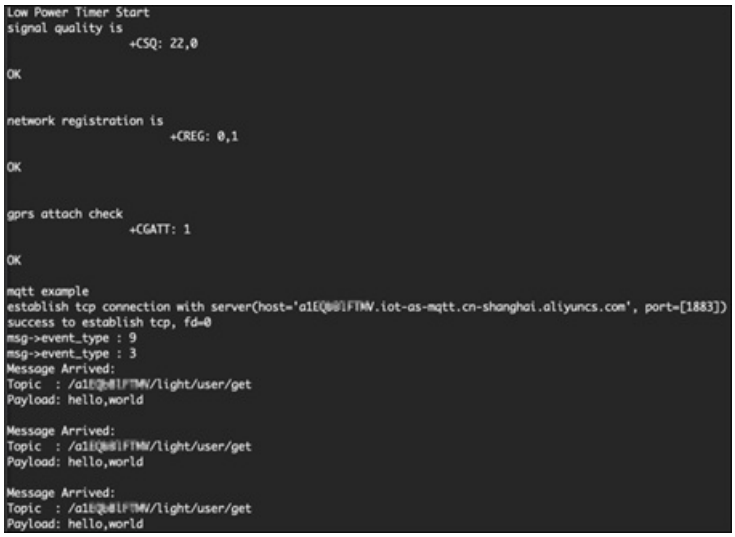
```
int HAL_AT_CONN_Close(int fd, int32_t remote_port)
int HAL_AT_CONN_Deinit(void)
int HAL_AT_CONN_DomainToIp(char *domain, char ip[16])
int HAL_AT_CONN_Init(void)
int HAL_AT_CONN_Send(int fd, uint8_t *data, uint32_t len, char remote_ip[16], int32
_t remote_port, int32_t timeout)
int HAL_AT_CONN_Start(at_conn_t *conn)
int32_t HAL_AT_Uart_Deinit(uart_dev_t *uart)
int32_t HAL_AT_Uart_Init(uart_dev_t *uart)
int32_t HAL_AT_Uart_Recv(uart_dev_t *uart, void *data, uint32_t expect_size, uint32
_t *recv_size, uint32_t timeout)
int32_t HAL_AT_Uart_Send(uart_dev_t *uart, const void *data, uint32_t size, uint32_
t timeout)
int HAL_GetDeviceName(char device_name[IOTX_DEVICE_NAME_LEN + 1])
int HAL_GetDeviceSecret(char device_secret[IOTX_DEVICE_SECRET_LEN + 1])
int HAL_GetFirmwareVersion(char *version)
int HAL_GetProductKey(char product_key[IOTX_PRODUCT_KEY_LEN + 1])
void *HAL_Malloc(uint32_t size);
void HAL_Free(void *ptr);
void *HAL_MutexCreate(void)
void HAL_MutexDestroy(void *mutex)
void HAL_MutexLock(void *mutex)
void HAL_MutexUnlock(void *mutex)
void HAL_Printf(const char *fmt, ...)
void HAL_SleepMs(uint32_t ms)
int HAL_Snprintf(char *str, const int len, const char *fmt, ...)
uint64_t HAL_UptimeMs(void)
```

4. Add the file to the IAR project.



5. Run the project and test data communication.

   After the test is successful, the device logs are displayed, as shown in the following figure.



   In the IoT Platform console, choose **Maintenance > Device Log** to view logs generated when the device reports data to IoT Platform.

1.11. Connect a serial device to IoT Platform by using a DTU device

1.11.1. Overview

IoT Platform can communicate with serial devices. You can use data transmission unit (DTU) devices to connect serial devices to IoT Platform without changing communication protocols for these serial devices.

Scenarios

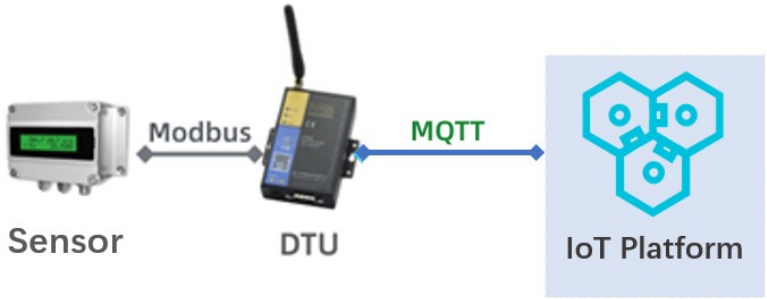
Devices for serial communication are used in a variety of scenarios, such as manufacturing, agriculture, healthcare, city, building, and industrial parks. Before you connect these devices to IoT Platform, you must enable communication between these devices and IoT Platform. However, the serial communication protocols used by these devices cannot be changed to communicate with IoT Platform. Therefore, data can be parsed only on the cloud. To connect these devices to IoT Platform, Alibaba Cloud works with hardware partners to provide a solution to communication between the serial devices and IoT Platform. This solution allows you to use DTU devices as proxies to enable communication between serial devices and IoT Platform by using simple configurations.

This topic uses a motor speed controller as an example to describe how to connect a serial device with IoT Platform by using a DTU device. This device complies with the integration standard of IoT Platform.

Implementation

To connect a serial device to IoT Platform, specify the certificate information of the serial device in a DTU device. Use the DTU device as a proxy for communication between the serial device and IoT Platform. Connect the serial device to the DTU device by using a serial port. Connect the DTU device to IoT Platform over the 2G, 3G, 4G, 5G, or Ethernet network. Use the DTU device as a proxy to enable communication between the serial device and IoT Platform.

The following figure shows how to connect a serial device to IoT Platform by using a DTU device.



Procedure

1. Configure a product and a device.
2. Configure a DTU device.
3. Test data communication.

1.11.2. Configure a product and a device

This article describes how to create a product whose data type is custom, create a device for the product, obtain the device certificate, define a Thing Specification Language (TSL) model, and then write a data parsing script. After you perform these operations, you can connect the device to IoT Platform by using a data transmission unit (DTU) device.

Create a product and a device

In the IoT Platform console, create a product and a device, and obtain the device certificate. The device certificate consists of a ProductKey, a DeviceName, and a DeviceSecret.

1. Log on to the IoT Platform console.

2.

3. In the left-side navigation pane, choose **Devices > Products**. On the Products page, click **Create Product** and create a product.

   Product information

Parameter	Description
Product	The name of the product. Example: **VariableFrequencyMotor**.
Category	Select **Custom Category**.
Node Type	Select **Directly Connected Device**.
Network Connection Method	Select **Cellular (2G / 3G / 4G / 5G)**.
Data Type	Select **Custom**.
Authorization Mode	Select **Device Secret**.

4. In the left-side navigation pane, choose **Devices** > Devices. On the Devices page, click **Add Device** and create a device for the product.
   After the device is created, go to the **Devices** page. Find the created device and click **View** in the Actions column. On the **Device Details** page, view the information about the device certificate. The device certificate will be configured to the DTU device. Keep it properly.

Define a TSL model

In this example, the RotateSpeed, Current, and RotateSpeedSet properties are defined for a variable frequency motor. For more information about TSL-related concepts, see What is a TSL model?.

1. Log on to the IoT Platform console.In the left-side navigation pane, choose **Devices > Products**.
2. On the **Products** page, find the created product and click **View** in the Actions column.
3. On the **Product Details** page, click the **Define Feature** tab, and then click **Edit Draft**. On the Edit Draft page, click **Add Self-defined Feature**.
4. Create the properties that are described in the following table. Then, click **Release online** to publish the TSL model.

Feature Type	Feature Name	Identifier	Data Type	Value Range	Step	Unit	Read/Write Type
Properties	RotateSpeed	speed	int32	0 ~ 3000	1	r/min	Read-only
Properties	Current	current	int32	0 ~ 300	1	A	Read-only
Properties	RotateSpeedSet	setspeed	int32	0 ~ 3000	1	r/min	Read/Write

Write a data parsing script

IoT Platform processes data of the Alink JSON format. If devices communicate with IoT Platform by using DTU devices, IoT Platform cannot process pass-through data that is sent from the devices.

IoT Platform provides the data parsing feature. This feature allows you to convert the custom format of upstream data into the Alink JSON format and the format of downstream data into the custom format of your device. To use the data parsing feature, you must write a data parsing script in the IoT Platform console and submit the script to IoT Platform. You must write a data parsing script based on upstream and downstream data.

1. On the **Product Details** page, click the **Data Parsing** tab.
2. On this tab, set the **Script Language** parameter next to **Edit Script** to **JavaScript(ECMAScript 5)**. Then, enter a data parsing script in the code editor.

   For more information about how to write a data parsing script, see Submit a data parsing script.

   In this example, a device sends data of the hexadecimal format to IoT Platform. Therefore, you must use a data parsing script to convert upstream data of the hexadecimal format into data of the Alink JSON format and downstream data of the Alink JSON format into data of the hexadecimal format.

   The following code provides an example on how to write a data parsing script:

```
var ALINK_ID = "12345";
var ALINK_VERSION = "1.1";
var ALINK_PROP_POST_METHOD = 'thing.event.property.post';
// var ALINK_EVENT_TEMPERR_METHOD = 'thing.event.TempError.post';
// var ALINK_EVENT_HUMIERR_METHOD = 'thing.event.HumiError.post';
var ALINK_PROP_SET_METHOD = 'thing.service.property.set';
// var ALINK_SERVICE_THSET_METHOD = 'thing.service.SetTempHumiThreshold';
/* * * * * *
* Upstream data:
* 0102 // Two bytes in total * Results:
* {"method":"thing.event.TempError.post","id":"12345","params":{"Temperature": 2},"vers
ion":"1.1"}
* Upstream data:
* 0202 // Two bytes in total * Results:
* {"method":"thing.event.HumiError.post","id":"12345","params":{"Humidity":2}, "version
":"1.1"}
*/
/* The rawDataToProtocol() function converts upstream data that is submitted by a devic
e into TSL data of the Alink JSON format. */
function rawDataToProtocol(bytes) {
    /* Convert upstream data into an array. Upstream data is stored in the bytes object
. */
    var uint8Array = new Uint8Array(bytes.length);
    for (var i = 0; i < bytes.length; i++) {
        uint8Array[i] = bytes[i] & 0xff;
    }
    var params = {};                            // Create an object to store device pro
perties.
    var jsonMap = {};                           // Create an object to define the forma
t of an Alink message.
    /* Specify header fields for an Alink message. */
    jsonMap['version'] = ALINK_VERSION;         // The version number of the Alink prot
ocol.
    jsonMap['id'] = ALINK_ID;                   // The ID of the Alink message.
```

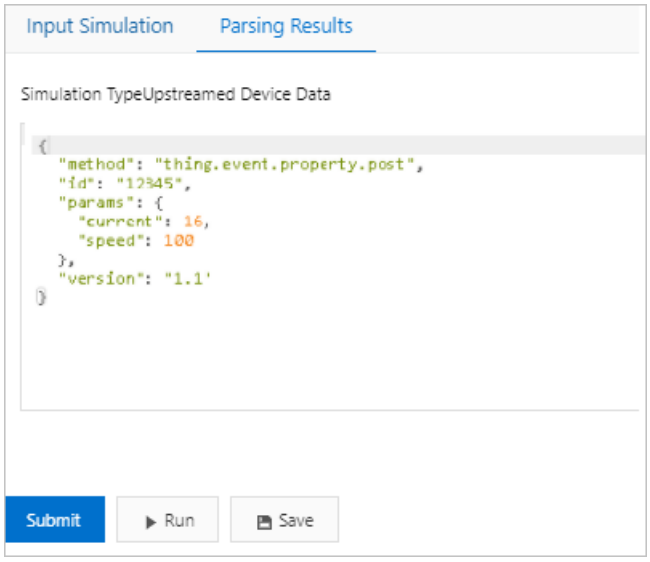
```
    jsonMap['method'] = ALINK_PROP_POST_METHOD; // The method that is used to submit de
vice properties.
    /* Specify properties for the Alink message. */
    params['speed'] = uint8Array[0];            // Convert the first received byte into
the value of the RotateSpeed property.
    params['current'] = uint8Array[1];          // Convert the second received byte int
o the value of the Current property.
    jsonMap['params'] = params;                 // Pack the required parameters to a JS
ON array.
    return jsonMap;                             // Return the results to IoT Platform.
}
// The following code describes helper functions:
function buffer_uint8(value)
{
    var uint8Array = new Uint8Array(1);
    var dv = new DataView(uint8Array.buffer, 0);
    dv.setUint8(0, value);
    return [].slice.call(uint8Array);
}
function buffer_int16(value)
{
    var uint8Array = new Uint8Array(2);
    var dv = new DataView(uint8Array.buffer, 0);
    dv.setInt16(0, value);
    return [].slice.call(uint8Array);
}
function buffer_int32(value)
{
    var uint8Array = new Uint8Array(4);
    var dv = new DataView(uint8Array.buffer, 0);
    dv.setInt32(0, value);
    return [].slice.call(uint8Array);
}
function buffer_float32(value)
{
    var uint8Array = new Uint8Array(4);
    var dv = new DataView(uint8Array.buffer, 0);
    dv.setFloat32(0, value);
    return [].slice.call(uint8Array);
}
/* The protocolToRawData() function converts downstream data into hexadecimal data that
can be recognized by the device. */
function protocolToRawData(json)
{
    var method = json['method'];
    var id = json['id'];
    var version = json['version'];
    var payloadArray = [];
    if (method == ALINK_PROP_SET_METHOD)    // If the device receives the Set Device Pr
operties command from IoT Platform, the following code is run:
    {
        var send_params = json['params'];
        var prop_cur = send_params['setspeed'];   // Specify the received value for the
prop_cur parameter.
```

```
        // Concatenate raw data based on the custom format.
        payloadArray = payloadArray.concat(buffer_uint8(0x55)); // The first byte that
indicates the header of a message.
        payloadArray = payloadArray.concat(buffer_uint8(prop_bool)); // The second byte
that indicates the value of a property.
    }
    return payloadArray;    // Send values to the device.
}
function transformPayload(topic, rawData) {
    var jsonObj = {};
    return jsonObj;
}
```

3. Test the script.
   - Parse data that is submitted by a device.
     a. Select **Upstreamed Device Data** from the Simulation Type drop-down list.
     b. On the **Input Simulation** tab, enter test data in the field.

        The preceding script specifies the first byte as the value of the RotateSpeed property and the second byte as the Current property. If you enter 6410, the first byte 64 indicates that the rotating speed is 100 RPM and the second byte 10 indicates that the electric current is 16 A.
     c. Click **Run**.

        The **Parsing Results** tab displays the results, as shown in the following figure.



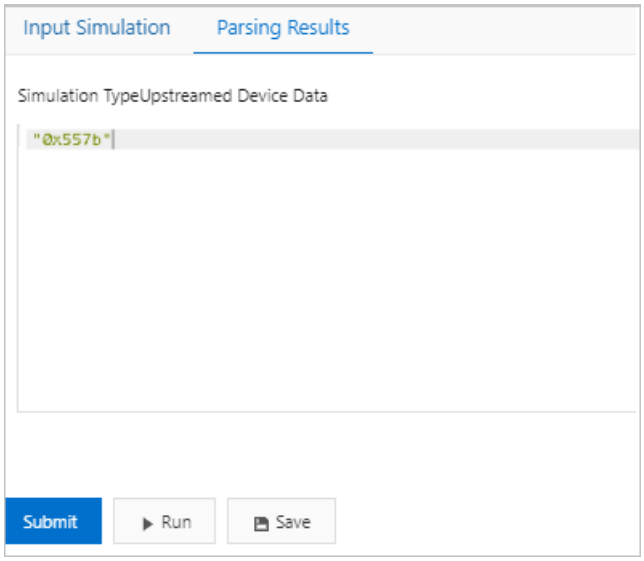
   - Parse data that is sent from IoT Platform.
     a. Select **Received Device Data** from the Simulation Type drop-down list.

b. On the **Input Simulation** tab, enter test data in the field. Sample downstream data:

```
{
    "method": "thing.service.property.set",
    "id": "12345",
    "version": "1.0",
    "params": {
        "setspeed": 123
    }
}
```

c. Click **Run**.

The **Parsing Results** tab displays the results, as shown in the following figure.



4. After you confirm that test data can be parsed by the script as expected, click **Submit** to submit the script to IoT Platform.

> **Note** IoT Platform cannot use scripts that are in the draft state. Before IoT Platform can use a script to parse data, you must submit the script.

What's next

Configure a DTU device

1.11.3. Configure a DTU device

This topic describes how to use a data transmission unit (DTU) configuration tool to configure a serial port for a DTU device, obtain the device certificate information, and connect the DTU device to IoT Platform. The certificate information includes the ProductKey, DeviceName, and DeviceSecret parameters.

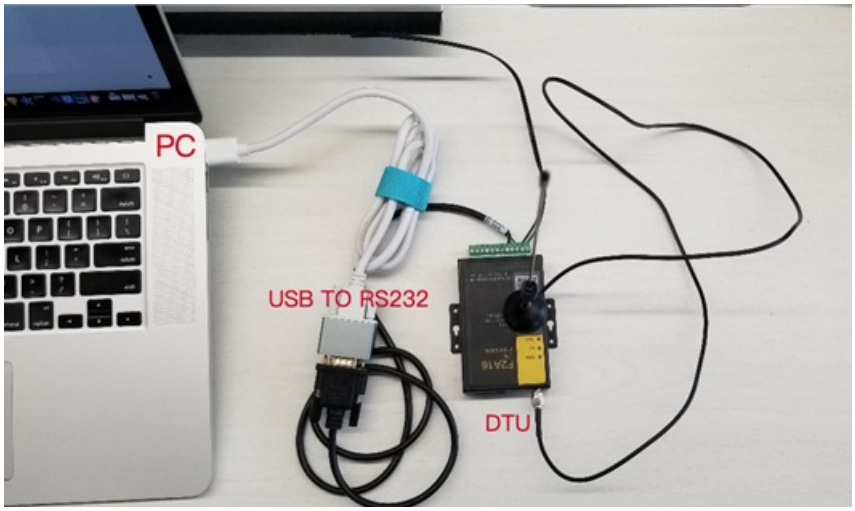
Context

This section uses an F2x16 DTU device as an example. You can use a PC to simulate the DTU device for data communication. Connect the DTU device to the PC by using a USB to serial adapter cable.

> **Note** Make sure that the DTU device is connected to the Internet.

Procedure

1. Connect the DTU device to a USB port of the PC.



2. On the PC, open the DTU configuration tool. This topic uses an F2x16 configuration tool as an example.
3. Set a valid port number, set a baud rate, and open a serial port.
4. Click **login configuration** to bring DTU into the configuration state.
5. Click **Read Configuration** to obtain the current settings of the DTU device.
6. On the right side of the **Configuration** window, click **Serial Port** to configure a serial port.

> **Note** You must select **PORT** in the Protocol field.

7. Click **IoT Integration** and specify the certificate information of the device and the name of the region where the device resides. You can obtain the information from the IoT Platform console.
8. Click **Apply** to apply the settings.

   If you fail to apply the settings, click **Exit** and try again later.
9. After the configuration is complete, click **Exit** to switch the DTS device to the normal mode.
10. Power off and then power on the DTU device. If the Online LED of the DTU device lights up, it indicates that the DTU device is connected to IoT Platform.

    You can also view the status of the DTU device in the IoT Platform console.

What's next

Test data communication

1.11.4. Test data communication

This topic describes how to use a data transmission unit (DTU) as a device to report data to IoT Platform and receive data from IoT Platform.

Test data reporting

1. Start a sierial port debugging tool. This topic uses sscom v4.2 as an example.

   > **Note** Before you use the debugging tool to simulate data sending and receiving, you must exit the DTU configuration tool.

2. Set the required parameters for the debugging tool, open a serial port, and then click **Send**.

   Set the spinning speed and electric current parameters based on the TSL model you created in IoT Platform. Then, send the values of the parameters to IoT Platform. Set the spinning speed to 150, electric current to 10 A. In the debugging tool, enter the following hex digits in sequence: 96 and 0A.

3. After data is sent, turn on the Real-time Refresh switch.
   i. Log on to the IoT Platform console. In the left-side navigation pane, choose **Devices > Devices**.
   ii. Find the target device and click **View**.
   iii. On the **Device Details** page, choose **TSL Data > Status**, and turn on the **Real-time Refresh** switch.
   Wait a few minutes until the uploaded data is displayed.

Test data receiving

On the **Debug Physical Device** tab, send a command to set the spinning speed, and test data receiving for the DTU device.

1. Log on to the IoT Platform console. In the left-side navigation pane, choose **Maintenance > Online Debug**.
2. Select the device that you want to debug and click the **Debug Physical Device** tab.
3. Select the spinning speed property in the Debug Feature field, select **Set** in the Method field, enter a value, and then click **Send Command**.
4. After the command is sent, view the received data in the debugging tool for the DTU device.

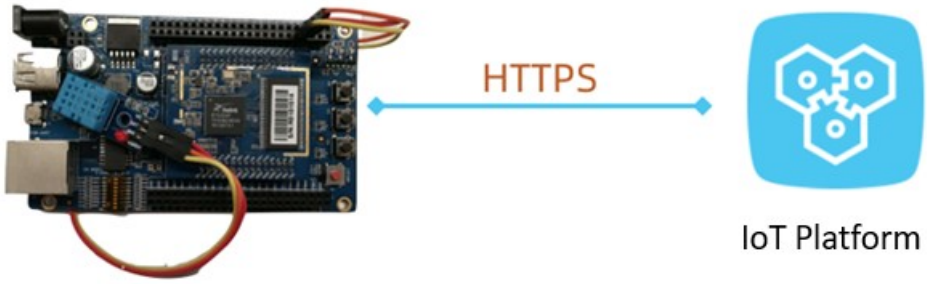
If data is displayed as expected in both the IoT Platform console and the device, it indicates a successful configuration.

1.12. Connect environmental sensors to IoT Platform by using HTTPS

Devices can be connected over HTTPS to IoT Platform only in the China (Shanghai) region. The HTTPS communication method is available only for scenarios where devices report data to IoT Platform. Only the POST request method is supported. A device can report a maximum of 128 KB of data at a time.

Context

This topic describes how to enable communication between a device and IoT Platform by using HTTPS and report data from the device to IoT Platform. It uses an environmental sensor as an example.



Create a product and add a device

In the IoT Platform console, create a product, add a device to the product, create a Thing Specification Language (TSL) model, and obtain certificate information about the device. The certificate information includes the ProductKey, DeviceName, and DeviceSecret parameters.

1. Log on to the IoT Platform console.
2. In the left-side navigation pane, choose **Devices > Products**. On the Products page, click **Create Product** to create a product.

Parameter	Description
Product Name	The name of the product.
[DO NOT TRANSLATE]	[DO NOT TRANSLATE]
Node Type	Select **Directly Connected Device**.
Network Connection Method	Select **Wi-Fi**.
Data Type	Select **ICA Standard Data Format (Alink JSON)**.
Authentication Mode	Select **Device Secret**.

3. After the product is created, click **Create TSL**.
4. On the **Define Feature** tab of the **Product Details** page, choose **Edit Draft > Add Self-defined Feature** to add properties.

   The environmental sensor will report temperature data and humidity data. Therefore, you must add the following properties.

Feature type	Feature name	Identifier	Data type	Value range	Step	Read/write type
Property	Temperature	temperature	int32	-10 to 50	1	Read-only
Property	Humidity	humidity	int32	1 to 100	1	Read-only

5. After the TSL model is created, click **Release Online** to publish the TSL model.
6. In the left-side navigation pane, click **Devices** and click **Add Device** to add a device to the

product.
After the device is added, obtain the values of ProductKey, DeviceName, and DeviceSecret parameters.

Send data from a device to a topic

Use HTTPS to enable communication between a device and IoT Platform and use POST requests to report temperature data and humidity data.

1. Obtain a device token.

   When a device attempts to communicate with IoT Platform, IoT will authenticate the device. After the authentication is complete, a device token is returned. The token is required when the device reports data to IoT Platform.

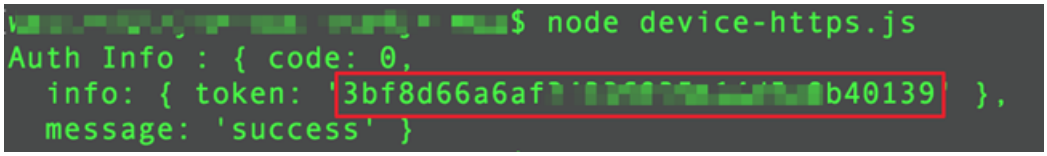
   The following table lists the parameters that you must specify to obtain a device token.

Parameter	Description
method	The request method. You must specify *POST* for the parameter.
uri	Specify *https://iot-as-http.cn-shanghai.aliyuncs.com/auth*.
productKey	The product key. You can obtain the information from the **Device Details** page of the IoT Platform console.
deviceName	The name of the device. You can obtain the information from the **Device Details** page of the IoT platform console.
clientId	The client ID. The ID can be a maximum of 64 characters in length. The ID can be the MAC address or SN of the device. The following sample code uses the random() function to generate a random ID.
timestamp	The timestamp. The following sample code uses the now() function to obtain a timestamp.
signmethod	The type of the algorithm. Valid values: hmacmd5 and hmacsha1.
sign	The signature. Use the following function to generate a signature. `password = signHmacSha1(params, deviceConfig.deviceSecret)`

   Use the following code to obtain a device token.

```
var rp = require('request-promise');
const crypto = require('crypto');
const deviceConfig = {
    productKey: "<yourProductKey>",
    deviceName: "<yourDeviceName>",
    deviceSecret: "<yourDeviceSecret>"
}
//Obtain a token.
rp(getAuthOptions(deviceConfig))
    .then(function(parsedBody) {
        console.log('Auth Info :',parsedBody)
    })
    }).catch(function (err) {
        console.log('Auth err :'+JSON.stringify(err))
    });
//Specify the required parameters for authentication.
function getAuthOptions(deviceConfig) {
    const params = {
        productKey: deviceConfig.productKey,
        deviceName: deviceConfig.deviceName,
        timestamp: Date.now(),
        clientId: Math.random().toString(36).substr(2),
    }
    //Specify the required parameters.
    var password = signHmacSha1(params, deviceConfig.deviceSecret);
    var options = {
        method: 'POST',
        uri: 'https://iot-as-http.cn-shanghai.aliyuncs.com/auth',
        "body": {
            "version": "default",
            "clientId": params.clientId,
            "signmethod": "hmacSha1",
            "sign": password,
            "productKey": deviceConfig.productKey,
            "deviceName": deviceConfig.deviceName,
            "timestamp": params.timestamp
        },
        json: true
    };
    return options;
}
//HmacSha1 sign
function signHmacSha1(params, deviceSecret) {
    let keys = Object.keys(params).sort();
    // Sort parameters in the alphabetical order.
    keys = keys.sort();
    const list = [];
    keys.map((key) => {
        list.push(`${key}${params[key]}`);
    });
    const contentStr = list.join('');
    return crypto.createHmac('sha1', deviceSecret).update(contentStr).digest('hex');
}
```

After you specify the required parameters in the code, run the code to initiate device authentication. If the authentication is successful, a device token is returned.



> **Note** A device token will expire after seven days. Make sure that you have solutions to token expiration issues.

2. Report data from the device.

   After a device is authenticated, a token is returned. You can specify the token for the password parameter that is required for data reporting.

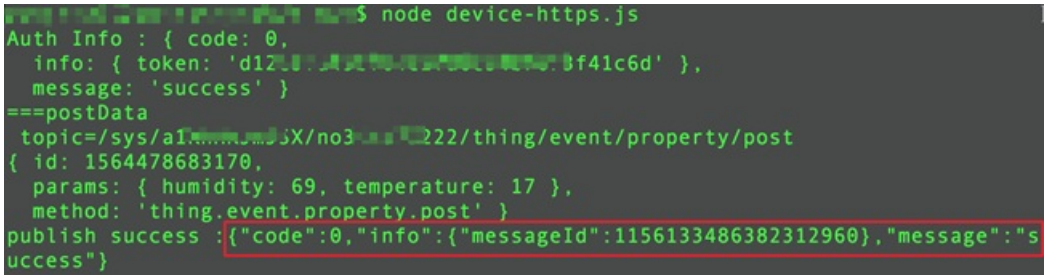
   The following table lists the parameters that you must specify to report data.

Parameter	Description
method	The request method. You must specify *POST*.
uri	The syntax is `https://iot-as-http.cn-shanghai.aliyuncs.com/topic + topic`. The URL consists of the HTTP endpoint of IoT Platform and the name of a topic. The second topic specifies the name of a topic of the following syntax: `/sys/${deviceConfig.productKey}/${deviceConfig.deviceName}/thing/event/property/post`
body	The data to report.
password	The device token.
Content-Type	The content type of the data. Set the value to application/octet-stream.

   The following code shows how to report data.

```
const topic = `/sys/${deviceConfig.productKey}/${deviceConfig.deviceName}/thing/event/p
roperty/post`;
//Report data.
pubData(topic, token, getPostData())
function pubData(topic, token, data) {
    const options = {
        method: 'POST',
        uri: 'https://iot-as-http.cn-shanghai.aliyuncs.com/topic' + topic,
        body: data,
        headers: {
            password: token,
            'Content-Type': 'application/octet-stream'
        }
    }
    rp(options)
        .then(function(parsedBody) {
            console.log('publish success :' + parsedBody)
        })
        .catch(function(err) {
            console.log('publish err ' + JSON.stringify(err))
        });
}
// Create test data that conforms to the TSL model.
function getPostData() {
    var payloadJson = {
        id: Date.now(),
        params: {
            humidity: Math.floor((Math.random() * 20) + 60),
            temperature: Math.floor((Math.random() * 20) + 10)
        },
        method: "thing.event.property.post"
    }
    console.log("===postData\n topic=" + topic)
    console.log(payloadJson)
    return JSON.stringify(payloadJson);
}
```

After you specify the required parameters in the code, run the code to report data. Then, view the results in local logs.



To find the temperature data and humidity data that are reported from the device, follow these steps: Log on to the IoT Platform console. Go to the **Device Details** page. View the data on the **Status** tab. If data is displayed as expected, it indicates that the device is connected to IoT Platform and the data is reported.

For more information about HTTPS communication, see Establish connections over HTTPS.

1.13. Connect a Linux device to IoT Platform

You can directly compile and run the demo in the device SDK for C on a Linux device to connect to IoT Platform by using Message Queuing Telemetry Transport (MQTT). This article describes how to configure and connect a Linux device to IoT Platform. In the following example, the demo in the device SDK for C is compiled on a device that runs the Ubuntu x86_64 operating system.

Context

For more information about the device SDK for C, see Overview.

Create a product and a device

1. Log on to the IoT Platform console.
2.
3. In the left-side navigation pane, choose **Devices > Products**. On the Products page, click **Create Product** to create a product.

Parameter	Description
Product Name	The name of the product.
Node Type	Select **Directly Connected Device**.
Network Connection Method	Select **Wi-Fi**.
Data Type	Select **ICA Standard Data Format (Alink JSON)**.
Authorization Mode	Select **Device Secret**.

4. In the left-side navigation pane, choose **Devices** > **Devices**. On the **Devices** page, click **Add Device** to add a device to the product.
   After the device is added, obtain the certificate information of the device. The certificate information includes the ProductKey, DeviceName, and DeviceSecret.

Create a TSL model

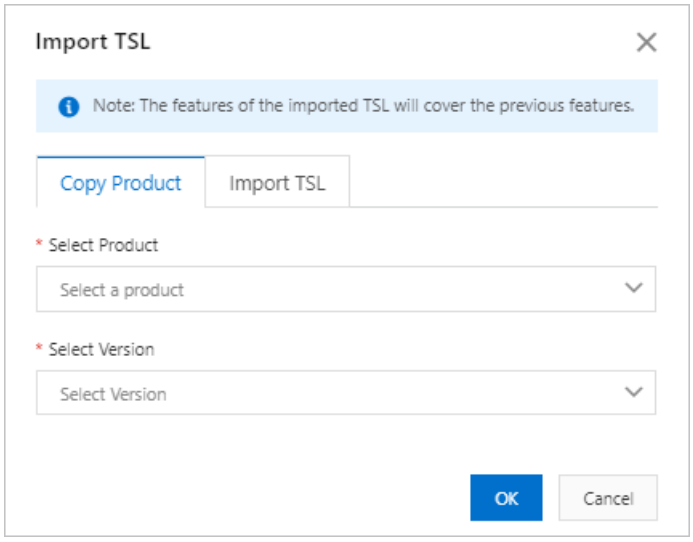
In the package of the device SDK for C provided by IoT Platform, you can obtain a JSON file that contains a complete sample Thing Specification Language (TSL) model. In the following example, the JSON file is imported to apply the TSL model to the product.

1. Edit the JSON file.
   i. Download the package of the device SDK for C.
   ii. Decompress the package and open the *model_for_examples.json* file in the *src/dev_model/examples* directory.

iii. Specify the value of the ProductKey that you obtain from IoT Platform for the productKey parameter in the JSON file. Then, save the file.

```
{
    "schema":"https://iotx-tsl.oss-ap-southeast-1.aliyuncs.com/schema.json",
    "profile":{
        "productKey":"a1pkc******"
    },
    "services":[
        {
            "outputData":[

            ],
            "identifier":"set",
            "inputData":[
                {
                    "identifier":"PowerSwitch",
                    "dataType":{
                        "specs":{
                            "0":"on",
                            "1":"off"
                        },
                        "type":"bool"
                    },
                    "name":"PowerSwitch"
                }
            ],
            "method":"thing.service.property.set",
            "name":"set",
            "required":true,
            "callType":"async",
            "desc":"PropertySettings"
        },
```

2. Log on to the IoT Platform console. On the **Products** page, find the product that you created earlier and click **View** in the Actions column.
3. On the **Define Feature** tab of the **Product Details** page, click **Edit Draft** and then **Import**.
4. In the Import TSL dialog box, click the **Import TSL** tab, upload the JSON file that you edited in the preceding step, and then click **OK**.



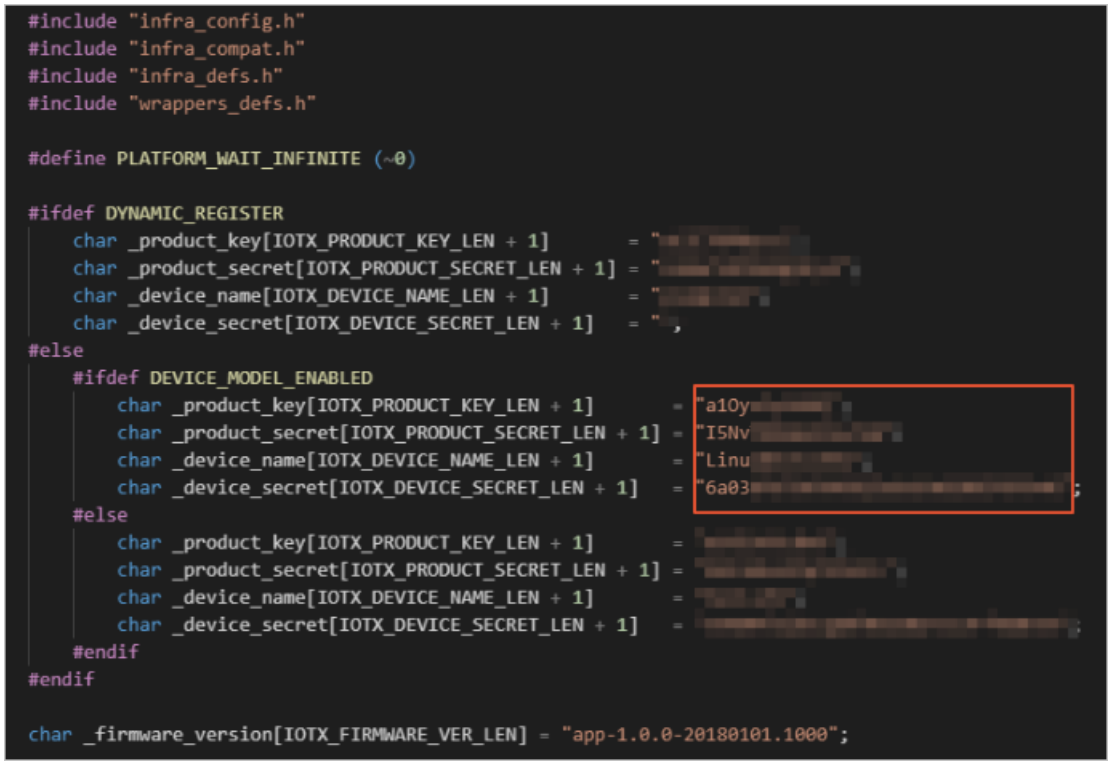
After the file is imported, all features that are defined in the JSON file appear on the Edit Draft page.

5. Click **Release online** to publish the TSL model.

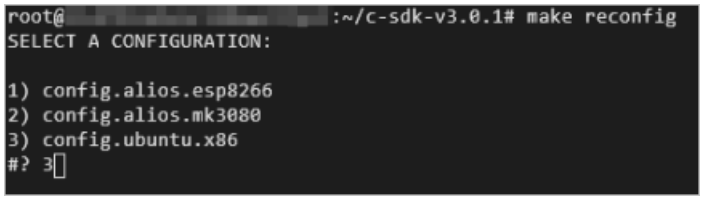
Configure the SDK demo

Copy the package of the device SDK for C to the Linux device. Decompress the package and modify the device information in a configuration file.

1. In the *wrappers/os/ubuntu* directory, specify the certificate information of your device in the *HAL_OS_linux.c* file.



2. Compile the SDK demo. In the root directory of the SDK, run the **make reconfig** command, enter *3*, and then run the **make** command.



3. Run the SDK demo.

   In the root directory of the SDK, run the **./output/release/bin/linkkit-example-solo** command. The following figure shows the result.

```
root@                    :~/c-sdk-v3.0.1# ./output/release/bin/linkkit-example-solo
establish tcp connection with server(host='a1(      ).iot-as-mqtt.cn-shanghai.aliyuncs.com', port=[1883])
success to establish tcp, fd=3

> {
>     "id": "1",
>     "version": "1.0",
>     "params": [
>         {
>             "attrKey": "SYS_LP_SDK_VERSION",
>             "attrValue": "3.0.1",
>             "domain": "SYSTEM"
>         },
>         {
>             "attrKey": "SYS_SDK_LANGUAGE",
>             "attrValue": "C",
>             "domain": "SYSTEM"
>         }
>     ],
>     "method": "thing.deviceinfo.update"
> }
```

After the SDK demo is run, log on to the IoT Platform console. On the **Device Details** page, view the device status and the TSL data reported by the device.

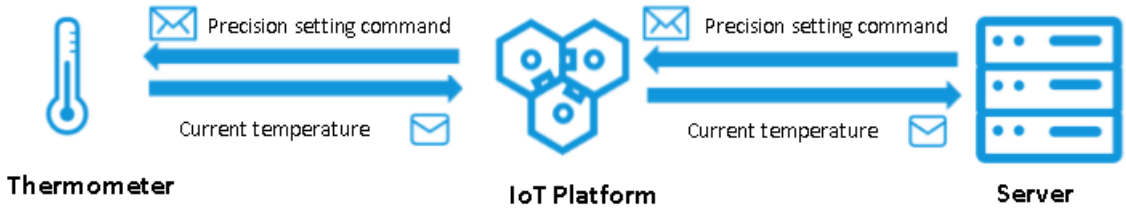
2.Communications

2.1. Use custom topics for communication

You can create custom topic categories in the IoT Platform console. Then, a device can send messages to a custom topic that belongs to a topic category. Your server can receive the messages by using an AMQP SDK. Your server can also call the Pub API operation to send commands to the device. Communication based on custom topics does not use the TSL model. In this case, you can define the data structure of the message.

Context

In this example, an electronic thermometer exchanges data with a server at a regular interval. The thermometer sends the real-time temperature data to the server, and the server sends the precision setting command to the thermometer.



Prepare the development environment

In this example, both devices and IoT Platform use SDKs for Java. You need to prepare the Java development environment first. You can download Java from the Java official website and install the Java development environment.

Create a product and a device

1. Log on to the IoT Platform console.
2. In the left-side navigation pane, choose **Devices > Products**.
3. Click **Create Product** to create a thermometer product.

   For more information, see Create a product.
4. After the thermometer product is created, find it on the Products page and click **View** in the Actions column.
5. On the **Products Details** page, click the **Topic Categories** tab. Then, you can create topic categories for the product.

   For more information, see Custom topics.

   In this example, you must define the following two topic categories:

   - /${productKey}/${deviceName}/user/devmsg: Devices send messages to this topic. Set Device Operation Authorizations to Publish for this topic category.
   - /${productKey}/${deviceName}/user/cloudmsg: Devices subscribe to this topic. Set Device

Operation Authorizations to Subscribe for this topic category.

6. Click the **Server-side Subscription** tab. Then, you can create an AMQP server-side subscription to subscribe to **Device Upstream Notification**.

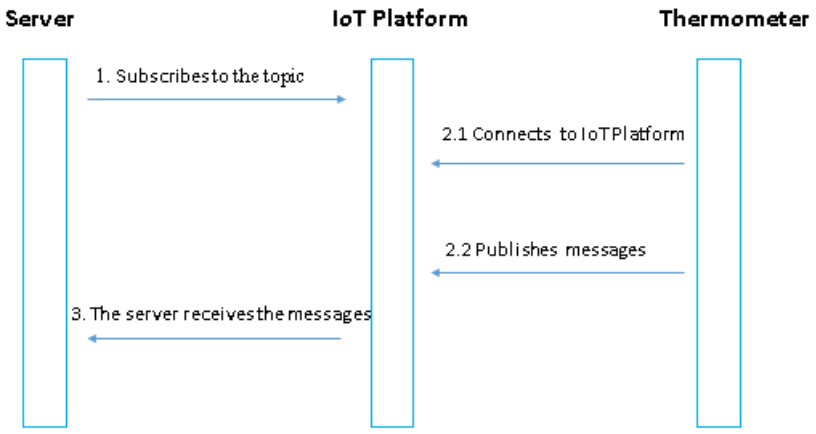
   For more information, see Configure an AMQP server-side subscription.

7. In the left-side navigation pane, choose **Devices > Devices**. Then, you can add a thermometer device to the thermometer product.

   For more information, see Create a device.

The server receives messages from the device

The following figure shows how the server receives messages from the device.



This section describes how to configure the server and the device to implement the process.

- The server receives messages from an AMQP client. Therefore, you must configure the AMQP client to connect the client with IoT Platform and enable the client to send messages.

  For more information, see Connect an AMQP client to IoT Platform.

  The following example shows how to connect a Qpid JMS 0.47.0 client as an AMQP client to IoT Platform.

  - Add the following Maven dependency:

```
<!-- amqp 1.0 qpid client -->
 <dependency>
   <groupId>org.apache.qpid</groupId>
   <artifactId>qpid-jms-client</artifactId>
   <version>0.57.0</version>
 </dependency>
 <!-- util for base64-->
 <dependency>
   <groupId>commons-codec</groupId>
  <artifactId>commons-codec</artifactId>
  <version>1.10</version>
</dependency>
```

- Connect the client to IoT Platform and enable the client to listen for device messages. Sample code:

- Configure the device SDK to connect the device with IoT Platform and enable the device to send messages.
  - Specify the device authentication parameters.

```
final String productKey = "XXXXXX";
final String deviceName = "XXXXXX";
final String deviceSecret = "XXXXXXXXX";
final String region = "XXXXXX";
```

  - Set the connection initialization parameters. These include the MQTT connection parameters, device parameters, TSL model parameters.

```
LinkKitInitParams params = new LinkKitInitParams();
// Set the MQTT connection parameters. Link Kit uses MQTT as the underlying protocol.
IoTMqttClientConfig config = new IoTMqttClientConfig();
config.productKey = productKey;
config.deviceName = deviceName;
config.deviceSecret = deviceSecret;
config.channelHost = productKey + ".iot-as-mqtt." + region + ".aliyuncs.com:1883";
// Set the device parameters.
DeviceInfo deviceInfo = new DeviceInfo();
deviceInfo.productKey = productKey;
deviceInfo.deviceName = deviceName;
deviceInfo.deviceSecret = deviceSecret;
// Register the initial status of the device.
Map<String, ValueWrapper> propertyValues = new HashMap<String, ValueWrapper>();
params.mqttClientConfig = config;
params.deviceInfo = deviceInfo;
params.propertyValues = propertyValues;
```

  - Initialize the connection.

```
// Initialize the connection and configure the callback function that is used after the
initialization succeeds.
LinkKit.getInstance().init(params, new ILinkKitConnectListener() {
     @Override
     public void onError(AError aError) {
         System.out.println("Init error:" + aError);
     }
     // Configure the callback function that is used after the initialization succeeds.
     @Override
     public void onInitDone(InitResult initResult) {
         System.out.println("Init done:" + initResult);
     }
 });
```

- Send a message from the device.

  After a device connects to IoT Platform, the device sends a message to the specified topic. Replace the content of the onInitDone function, as shown in the following example:

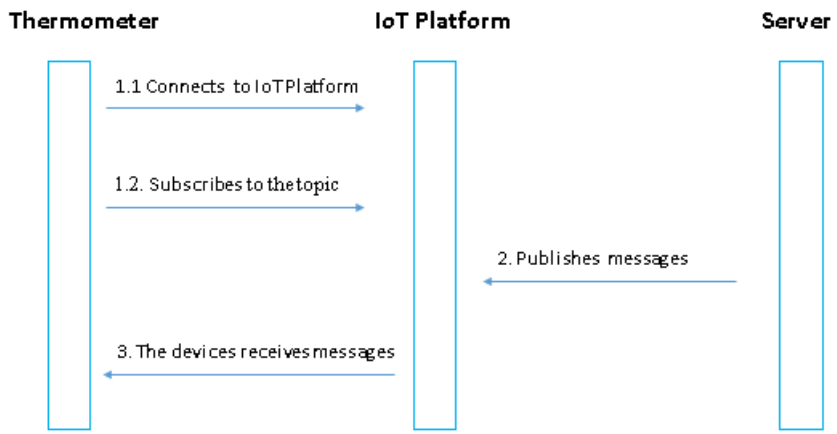
```
@Override
 public void onInitDone(InitResult initResult) {
     // Set the topic to which the message is published and the message content.
     MqttPublishRequest request = new MqttPublishRequest();
     request.topic = "/" + productKey + "/" + deviceName + "/user/devmsg";
     request.qos = 0;
     request.payloadObj = "{\"temperature\":35.0, \"time\":\"sometime\"}";
     // Publish the message and configure the callback functions that are used after th
e message is published.
     LinkKit.getInstance().publish(request, new IConnectSendListener() {
         @Override
         public void onResponse(ARequest aRequest, AResponse aResponse) {
             System.out.println("onResponse:" + aResponse.getData());
         }
         @Override
         public void onFailure(ARequest aRequest, AError aError) {
             System.out.println("onFailure:" + aError.getCode() + aError.getMsg());
         }
     });
 }
```

  The server receives the following message:

```
Message
{payload={"temperature":35.0, "time":"sometime"},
topic='/a1uzcH0****/device1/user/devmsg',
messageId='1131755639450642944',
qos=0,
generateTime=1558666546105}
```

The server sends messages to the device

The following figure shows how the server sends a message to the device.



- Configure the device SDK to subscribe to a topic.

For more information about how to specify device authentication parameters, set connection initialization parameters, and initialize the connection, see the sample code in the The server receives messages from the device section.

The device must subscribe to a specific topic before the device can receive messages sent by the server.

The following example shows how to configure the device SDK to subscribe to a topic:

```
// Configure the callback function that is used after the initialization succeeds.
@Override
public void onInitDone(InitResult initResult) {
    // Set the topic to which the device subscribes.
    MqttSubscribeRequest request = new MqttSubscribeRequest();
    request.topic = "/" + productKey + "/" + deviceName + "/user/cloudmsg";
    request.isSubscribe = true;
    // Send a subscription request and configure the callback functions that are used aft
er the subscription succeeds or fails.
    LinkKit.getInstance().subscribe(request, new IConnectSubscribeListener() {
        @Override
        public void onSuccess() {
            System.out.println("");
        }
        @Override
        public void onFailure(AError aError) {
        }
    });
    // Set the listener that listens for subscribed messages.
    IConnectNotifyListener notifyListener = new IConnectNotifyListener() {
        // Configure the callback functions that are used after a subscribed message is r
eceived.
        @Override
        public void onNotify(String connectId, String topic, AMessage aMessage) {
            System.out.println(
                "received message from " + topic + ":" + new String((byte[])aMessage.getD
ata()));
        }
        @Override
        public boolean shouldHandle(String s, String s1) {
            return false;
        }
        @Override
        public void onConnectStateChange(String s, ConnectState connectState) {
        }
    };
    LinkKit.getInstance().registerOnNotifyListener(notifyListener);
}
```

- Configure the IoT Platform SDK to enable IoT Platform to publish a message by calling the Pub API operation.

- Specify identity verification information.

```
String regionId = "XXXXXX";
String accessKey = "XXXXXX";
String accessSecret = "XXXXXXXXX";
final String productKey = "XXXXXX";
```

- Set the connection parameters.

```
// Set the parameters of the client.
DefaultProfile profile = DefaultProfile.getProfile(regionId, accessKey, accessSecret);
IAcsClient client = new DefaultAcsClient(profile);
```

- Set the parameters that are used to publish a message.

```
PubRequest request = new PubRequest();
request.setQos(0);
// Set the topic to which the message is published.
request.setTopicFullName("/" + productKey + "/" + deviceName + "/user/cloudmsg");
request.setProductKey(productKey);
// Set the MessageContent parameter. The message content must be encoded in Base64. Oth
erwise, the message content will appear as garbled characters.
request.setMessageContent(Base64.encode("{\"accuracy\":0.001,\"time\":now}"));
```

- Publish the message.

```
try {
     PubResponse response = client.getAcsResponse(request);
     System.out.println("pub success?:" + response.getSuccess());
 } catch (Exception e) {
     System.out.println(e);
 }
```

The device receives the following message:

```
msg = [{"accuracy":0.001,"time":now}]
```

Appendix: demo

You can Download the Pub/Sub demo to view the demo for this example.

For more information about how to connect an AMQP client to IoT Platform, see the following articles:

- Connect a client to IoT Platform by using the SDK for Java
- Connect a client to IoT Platform by using the SDK for .NET
- Connect a client to IoT Platform by using the SDK for Node.js
- Connect a client to IoT Platform by using the SDK for Python 2.7
- Connect a client to IoT Platform by using the SDK for PHP
- Connect a client to IoT Platform by using the SDK for Go

2.2. Remotely control a Raspberry Pi server

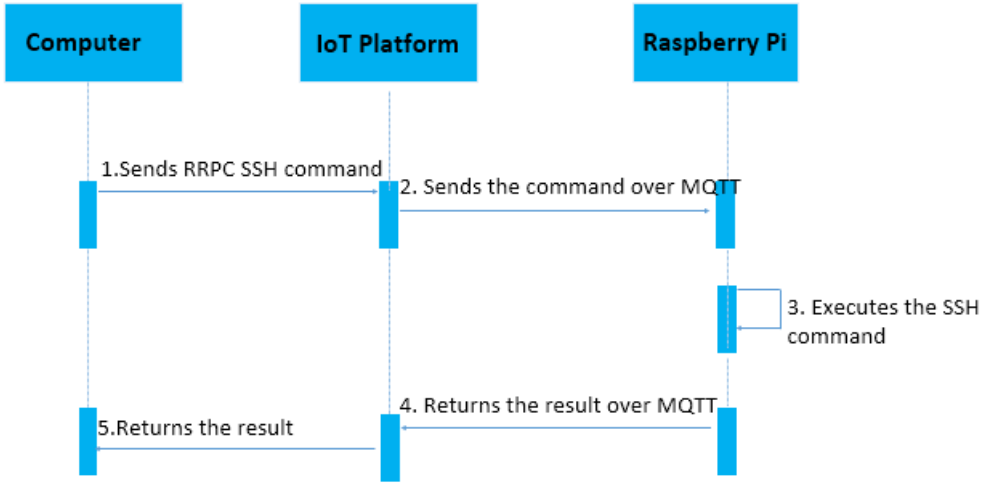
IoT Platform allows you to implement pseudo-Intranet penetration to remotely control a Raspberry Pi server that does not have a public IP address. This topic describes how to use IoT Platform to remotely control a Raspberry Pi server, and provides sample code.

Context

For example, you use Raspberry Pi to build a server in your company or at home to run some simple tasks, such as starting a script or downloading files. If the Raspberry Pi server does not have a public IP address and you are not in the company or at home, you cannot manage the server. If you use other Intranet penetration tools, disconnections may frequently occur. To resolve this issue, you can use the RRPC feature of IoT Platform together with the JSch library to remotely control the Raspberry Pi server.

> **Note** In this example, a product and a device in a public instance are used.

Process



The following process shows how to use IoT Platform to remotely control a Raspberry Pi server:

1. Call the IoT Platform RRPC operation on your computer to send a Secure Shell (SSH) command.
2. After IoT Platform receives the SSH command, IoT Platform sends the SSH command to the Raspberry Pi server over Message Queuing Telemetry Transport (MQTT).
3. The system runs the SSH command on the server.
4. The system encapsulates the result of the SSH command as an RRPC response on the server and sends the response to IoT Platform over MQTT.
5. IoT Platform sends the RRPC response to your computer.

> **Note** The RRPC timeout period is 5 seconds. If IoT Platform does not receive a response from the Raspberry Pi server within 5 seconds, a timeout error occurs. If you send a command that requires a long period of time to process, ignore the timeout error message.

Download SDKs and sample code

Before you can remotely control the Raspberry Pi server, develop an IoT Platform SDK and Link SDK.

- Install an IoT Platform SDK on your computer. You can use the Java sample code to develop the IoT

Platform SDK.

- Install a Link SDK on the Raspberry Pi server. You can use the Java sample code to develop the Link SDK.

The following sample code shows how to develop the IoT Platform SDK and Link SDK.

> **Note** The sample code supports only simple Linux commands, such as uname, touch, and mv. Complex commands, such as the commands that are used to modify files are not supported. To use complex commands, write code based on your business requirements.

Develop the Link SDK

After you install the Link SDK and download the sample code, add project dependencies and the required Java files.

The project can be exported as a JAR package and run on the Raspberry Pi server.

1. Add the following dependencies to the *pom.xml* file:

```
<! -- Link SDK -->
<dependency>
    <groupId>com.aliyun.alink.linksdk</groupId>
    <artifactId>iot-linkkit-java</artifactId>
    <version>1.1.0</version>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.8.1</version>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.40</version>
    <scope>compile</scope>
</dependency>
<! -- SSH client -->
<!-- https://mvnrepository.com/artifact/com.jcraft/jsch -->
<dependency>
    <groupId>com.jcraft</groupId>
    <artifactId>jsch</artifactId>
    <version>0.1.55</version>
</dependency>
```

2. Create a file named *SSHShell.java* to run SSH commands.

```
public class SSHShell {
    private String host;
    private String username;
    private String password;
    private int port;
    private Vector<String> stdout;
    public SSHShell(final String ipAddress, final String username, final String passwor
d, final int port) {
        this.host = ipAddress;
        this.username = username;
        this.password = password;
        this.port = port;
        this.stdout = new Vector<String>();
    }
    public int execute(final String command) {
        System.out.println("ssh command: " + command);
        int returnCode = 0;
        JSch jsch = new JSch();
        SSHUserInfo userInfo = new SSHUserInfo();
        try {
            Session session = jsch.getSession(username, host, port);
            session.setPassword(password);
            session.setUserInfo(userInfo);
            session.connect();
            Channel channel = session.openChannel("exec");
            ((ChannelExec) channel).setCommand(command);
            channel.setInputStream(null);
            BufferedReader input = new BufferedReader(new InputStreamReader(channel.get
InputStream()));
            channel.connect();
            String line = null;
            while ((line = input.readLine()) != null) {
                stdout.add(line);
            }
            input.close();
            if (channel.isClosed()) {
                returnCode = channel.getExitStatus();
            }
            channel.disconnect();
            session.disconnect();
        } catch (JSchException e) {
            e.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return returnCode;
    }
    public Vector<String> getStdout() {
        return stdout;
    }
}
```

3. Create a file named *SSHUserInfo.java* to verify the SSH username and password.

```
public class SSHUserInfo implements UserInfo {
    @Override
    public String getPassphrase() {
        return null;
    }
    @Override
    public String getPassword() {
        return null;
    }
    @Override
    public boolean promptPassphrase(final String arg0) {
        return false;
    }
    @Override
    public boolean promptPassword(final String arg0) {
        return false;
    }
    @Override
    public boolean promptYesNo(final String arg0) {
        if (arg0.contains("The authenticity of host")) {
            return true;
        }
        return false;
    }
    @Override
    public void showMessage(final String arg0) {
    }
}
```

4. Create a file named *Device.java* to establish an MQTT connection.

```
public class Device {
    /**
     * Establish a connection.
     *
     * @param productKey The ProductKey of the product.
     * @param deviceName The DeviceName of the device.
     * @param deviceSecret The DeviceSecret of the device.
     * @throws InterruptedException
     */
    public static void connect(String productKey, String deviceName, String deviceSecre
t) throws InterruptedException {
        // Initialize parameters.
        LinkKitInitParams params = new LinkKitInitParams();
        // Configure the required parameters to establish an MQTT connection.
        IoTMqttClientConfig config = new IoTMqttClientConfig();
        config.productKey = productKey;
        config.deviceName = deviceName;
        config.deviceSecret = deviceSecret;
        params.mqttClientConfig = config;
        // Specify the certificate information about the device.
        DeviceInfo deviceInfo = new DeviceInfo();
        deviceInfo.productKey = productKey;
        deviceInfo.deviceName = deviceName;
```

```
        deviceInfo.deviceSecret = deviceSecret;
        params.deviceInfo = deviceInfo;
        // Initialize a client.
        LinkKit.getInstance().init(params, new ILinkKitConnectListener() {
            public void onError(AError aError) {
                System.out.println("init failed !! code=" + aError.getCode() + ",msg="
+ aError.getMsg() + ",subCode="
                        + aError.getSubCode() + ",subMsg=" + aError.getSubMsg());
            }
            public void onInitDone(InitResult initResult) {
                System.out.println("init success !!");
            }
        });
        // Before you perform the following steps, make sure that the initialization is
complete. You can specify a reasonable value based on your business scenario.
        Thread.sleep(2000);
    }
    /**
     * Send a message.
     *
     @param topic The topic to which the message is sent.
     * @param payload The content of the message.
     */
    public static void publish(String topic, String payload) {
        MqttPublishRequest request = new MqttPublishRequest();
        request.topic = topic;
        request.payloadObj = payload;
        request.qos = 0;
        LinkKit.getInstance().getMqttClient().publish(request, new IConnectSendListener
() {
            @Override
            public void onResponse(ARequest aRequest, AResponse aResponse) {
            }
            @Override
            public void onFailure(ARequest aRequest, AError aError) {
            }
        });
    }
    /**
     * Subscribe to messages.
     *
     * @param topic The topic whose messages to which you want to subscribe.
     */
    public static void subscribe(String topic) {
        MqttSubscribeRequest request = new MqttSubscribeRequest();
        request.topic = topic;
        request.isSubscribe = true;
        LinkKit.getInstance().getMqttClient().subscribe(request, new IConnectSubscribeL
istener() {
            @Override
            public void onSuccess() {
            }
            @Override
            public void onFailure(AError aError) {
```

```
            }
        });
    }
    /**
     * Unsubscribe from messages.
     *
     * @param topic The topic whose messages from which you want to unsubscribe.
     */
    public static void unsubscribe(String topic) {
        MqttSubscribeRequest request = new MqttSubscribeRequest();
        request.topic = topic;
        request.isSubscribe = false;
        LinkKit.getInstance().getMqttClient().unsubscribe(request, new IConnectUnscribe
Listener() {
            @Override
            public void onSuccess() {
            }
            @Override
            public void onFailure(AError aError) {
            }
        });
    }
    /**
     * Terminate the connection.
     */
    public static void disconnect() {
        // Deinitialize the parameters.
        LinkKit.getInstance().deinit();
    }
}
```

5. Create a file named *SSHDevice.java*. The *SSHDevice.java* file includes the main method that is used to receive RRPC commands. You can also call `SSHShell` to run SSH commands, and return RRPC responses. In the *SSHDevice.java* file, you must enter the device certificate information and the SSH username and password. The device certificate information includes the ProductKey, DeviceName, and DeviceSecret.

```
public class SSHDevice {
    // ====================The start of the segment in which you configure the required
parameters.============================
    // The ProductKey of the product.
    private static String productKey = "";
    //
    private static String deviceName = "";
    // The DeviceSecret of the device.
    private static String deviceSecret = "";
    // The topic that you want to use for messaging. You can use the topic without the
need to create or define the topic.
    private static String rrpcTopic = "/sys/" + productKey + "/" + deviceName + "/rrpc/
request/+";
    // The endpoint or the IP address to which the SSH client needs to access.
    private static String host = "127.0.0.1";
    // The username that is used to log on to the SSH client.
    private static String username = "";
    // The password for the username.
```

```
    private static String password = "";
    // The number of the port that is used to log on to the SSH client.
    private static int port = 22;
    // ==================The end of the segment in which you configure the required pa
rameters.============================
    public static void main(String[] args) throws InterruptedException {
        // Listen to downstream data.
        registerNotifyListener();
        // Establish a connection.
        Device.connect(productKey, deviceName, deviceSecret);
        // Subscribe to a topic.
        Device.subscribe(rrpcTopic);
    }
    public static void registerNotifyListener() {
        LinkKit.getInstance().registerOnNotifyListener(new IConnectNotifyListener() {
            @Override
            public boolean shouldHandle(String connectId, String topic) {
                // Process messages only from specific topics.
                if (topic.contains("/rrpc/request/")) {
                    return true;
                } else {
                    return false;
                }
            }
            @Override
            public void onNotify(String connectId, String topic, AMessage aMessage) {
                // Receive RRPC requests and return RRPC responses.
                try {
                    // Run a remote command.
                    String payload = new String((byte[]) aMessage.getData(), "UTF-8");
                    SSHShell sshExecutor = new SSHShell(host, username, password, port)
;
                    sshExecutor.execute(payload);
                    // Obtain the output of the command.
                    StringBuffer sb = new StringBuffer();
                    Vector<String> stdout = sshExecutor.getStdout();
                    for (String str : stdout) {
                        sb.append(str);
                        sb.append("\n");
                    }
                    // Return the output to the server.
                    String response = topic.replace("/request/", "/response/");
                    Device.publish(response, sb.toString());
                } catch (UnsupportedEncodingException e) {
                    e.printStackTrace();
                }
            }
            @Override
            public void onConnectStateChange(String connectId, ConnectState connectStat
e) {
            }
        });
    }
}
```

Develop the IoT Platform SDK

After you install the IoT Platform SDK and download the sample code, add project dependencies and the required Java files.

1. Add the following dependencies to the *pom.xml* file:

   > **Notice** For more information about the latest version of the IoT Platform SDK, see Use IoT Platform SDK for Java.

   ```
   <! -- IoT Platform SDK -->
   <dependency>
       <groupId>com.aliyun</groupId>
       <artifactId>aliyun-java-sdk-iot</artifactId>
       <version>7.38.0</version>
   </dependency>
   <dependency>
       <groupId>com.aliyun</groupId>
       <artifactId>aliyun-java-sdk-core</artifactId>
       <version>4.5.6</version>
   </dependency>
   <!-- commons-codec -->
   <dependency>
       <groupId>commons-codec</groupId>
       <artifactId>commons-codec</artifactId>
       <version>1.8</version>
   </dependency>
   ```

2. Create a file named *OpenApiClient.java* to call IoT Platform API operations.

   ```
   public class OpenApiClient {
       private static DefaultAcsClient client = null;
       public static DefaultAcsClient getClient(String accessKeyID, String accessKeySecret
   ) {
           if (client != null) {
               return client;
           }
           try {
               IClientProfile profile = DefaultProfile.getProfile("cn-shanghai", accessKey
   ID, accessKeySecret);
               DefaultProfile.addEndpoint("cn-shanghai", "cn-shanghai", "Iot", "iot.cn-sha
   nghai.aliyuncs.com");
               client = new DefaultAcsClient(profile);
           } catch (Exception e) {
               System.out.println("create OpenAPI Client failed !! exception:" + e.getMess
   age());
           }
           return client;
       }
   }
   ```

3. Create a file named *SSHCommandSender.java*. The *SSHCommandSender.java* file includes the main method that is used to send SSH commands and receive responses to SSH commands. In the *SSHCo*

mmandSender.java file, you must specify the device certificate information, the AccessKey pair of your Alibaba Cloud account, and SSH commands. The device certificate information includes the ProductKey, DeviceName, and DeviceSecret.

```
public class SSHCommandSender {
    // ====================The start of the segment in which you configure the required
parameters.============================
    // The AccessKey ID of your Alibaba Cloud account.
    private static String accessKeyID = "";
    // The AccessKey secret of your Alibaba Cloud account.
    private static String accessKeySecret = "";
    // The ProductKey of the product.
    private static String productKey = "";
    // The DeviceName of the device.
    private static String deviceName = "";
    // ====================The end of the segment in which you configure the required pa
rameters.============================
    public static void main(String[] args) throws ServerException, ClientException, Uns
upportedEncodingException {
        // The remote Linux command.
        String payload = "uname -a";
        // Create an RRPC request.
        RRpcRequest request = new RRpcRequest();
        request.setProductKey(productKey);
        request.setDeviceName(deviceName);
        request.setRequestBase64Byte(Base64.encodeBase64String(payload.getBytes()));
        request.setTimeout(5000);
        // The client that is used to receive requests from the server.
        DefaultAcsClient client = OpenApiClient.getClient(accessKeyID, accessKeySecret)
;
        // Send an RRPC request.
        RRpcResponse response = (RRpcResponse) client.getAcsResponse(request);
        // Process RRPC responses.
        // The result that is returned after the execution of the response.getSuccess()
function indicates only that the RRPC request is sent. The returned response does not i
ndicate whether the request is received by the device or whether the device returned a
response based on the result.
        // You must check the result based on the RRPC response code that is returned.
For more information, see https://www.alibabacloud.com/help/doc-detail/69797.htm.
        if (response != null && "SUCCESS".equals(response.getRrpcCode())) {
            // The output.
            System.out.println(new String(Base64.decodeBase64(response.getPayloadBase64
Byte()), "UTF-8"));
        } else {
            // Failed to obtain the output and print the related RRPC response code.
            System.out.println(response.getRrpcCode());
        }
    }
}
```

2.3. Establish TSL-based communication

You can establish Thing Specification Language (TSL)-based communication between devices and IoT Platform over the Alink protocol. Devices submit properties or events to IoT Platform, and IoT Platform sends commands to devices to configure properties or call services. This topic provides sample Java code to describe how to establish TSL-based communication.

Prerequisites

- IoT Platform is activated.
- A Java development environment is installed.

Create a product and a device

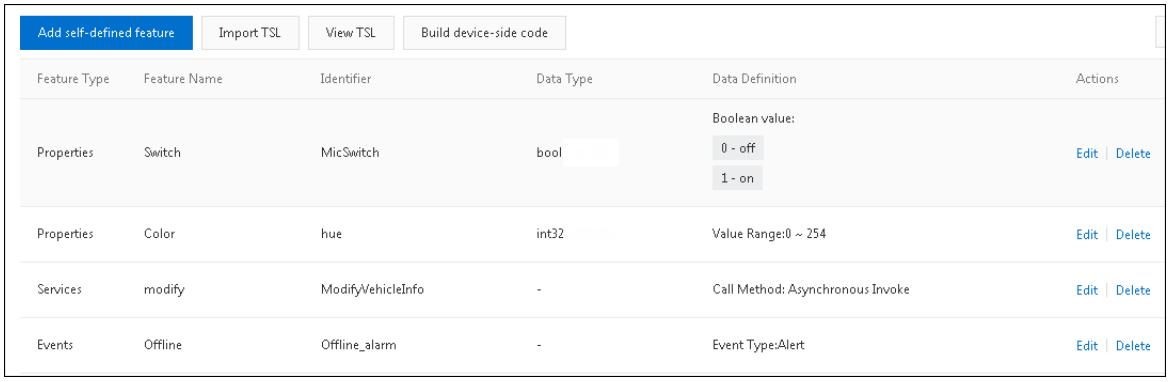
You must create a product and a device, and define TSL features for the product.

1. Log on to the IoT Platform console.
2.
3. In left-side navigation pane, choose **Devices > Products**.
4. On the Products page, click **Create Product**. On the Create Product page, specify a custom name for the product and select **Custom Category**. Use the default values for other parameters and click **OK**.

   For more information, see Create a product.
5. On the **Product Details** page of the product, click **Define Feature**. On the Define Feature tab, define a TSL model.

   In this example, an event, a service, and two properties are added to the **Default Module** section, as shown in the following figure.

   This topic provides a sample TSL model that you can import. For more information about how to import a TSL model, see Batch add TSL features.



6. In the left-side navigation pane, click **Devices**. On the Devices page, click Add Device to create a device.

   You can use the sample code to batch configure device properties and batch call device services. You must create at least two devices. For more information, see Create multiple devices at a time.

Download and install an SDK demo

The SDK demo includes the server-side sample code and device-side sample code.

1. Click iotx-api-demo to download the iotx-api-demo package and then decompress the package.
2. Start the Java development tool and import the decompressed *iotx-api-demo* folder.
3. Add the following Maven dependencies to the *pom.xml* file to import the IoT Platform SDK and a Link SDK:

```
<!-- https://mvnrepository.com/artifact/com.aliyun/aliyun-java-sdk-iot -->
<dependency>
    <groupId>com.aliyun</groupId>
    <artifactId>aliyun-java-sdk-iot</artifactId>
    <version>7.33.0</version>
</dependency>
<dependency>
    <groupId>com.aliyun</groupId>
    <artifactId>aliyun-java-sdk-core</artifactId>
    <version>3.5.1</version>
</dependency>
<dependency>
  <groupId>com.aliyun.alink.linksdk</groupId>
  <artifactId>iot-linkkit-java</artifactId>
  <version>1.2.0</version>
  <scope>compile</scope>
</dependency>
```

4. In the *java/src/main/resources/* directory, open the *config* file and specify the required information for initialization.

```
user.accessKeyID = <your accessKey ID>
user.accessKeySecret = <your accessKey Secret>
iot.regionId = <regionId>
iot.productCode = Iot
iot.domain = iot.<regionId>.aliyuncs.com
iot.version = 2018-01-20
```

Parameter	Description
accessKeyID	The AccessKey ID of your Alibaba Cloud account. To create or view your AccessKey pair, perform the following steps: Log on to the Alibaba Cloud Management console. Move the pointer over your profile picture, and click **AccessKey Management** to go to the **AccessKey Pair** page.
accessKeySecret	The AccessKey secret of your Alibaba Cloud account. You can obtain the AccessKey secret the same way you obtain the AccessKey ID.
regionId	The ID of the region where your IoT devices reside. For more information about region IDs, see Regions and zones.

Submit properties and events by using Link SDK

Configure Link SDK to connect to IoT Platform and submit properties and events.

The *ThingTemplate* file in the *java/src/main/com.aliyun.iot.api.common.deviceApi* directory contains the sample code.

- Specify the connection information.

  Replace the values of the productKey, deviceName, and deviceSecret parameters in the sample code with the information about your device certificate. Replace the value of the url parameter with your MQTT endpoint. For more information about how to obtain an endpoint, see View the endpoint of an instance. The endpoint must include port 1883.

```
public static void main(String[] args) {
        /**
         * The information about the device certificate.
         */
        String productKey = "your productKey";
        String deviceName = "your deviceName";
        String deviceSecret = "your deviceSecret";
        /*TODO: Replace the following endpoint with the endpoint of your instance.
       */
        String  url = "iot-6d***ql.mqtt.iothub.aliyuncs.com:1883";
        /**
         * The information about the MQTT connection.
         */
        ThingTemplate manager = new ThingTemplate();
        DeviceInfo deviceInfo = new DeviceInfo();
        deviceInfo.productKey = productKey;
        deviceInfo.deviceName = deviceName;
        deviceInfo.deviceSecret = deviceSecret;
        /**
         * The Java HTTP client supports TSLv1.2.
         */
        System.setProperty("https.protocols", "TLSv2");
        manager.init(deviceInfo, url);
    }
```

- Initialize the connection parameters.

```
public void init(final DeviceInfo deviceInfo, String url) {
        LinkKitInitParams params = new LinkKitInitParams();
        /**
         * Configure the parameters for MQTT initialization.
         */
        IoTMqttClientConfig config = new IoTMqttClientConfig();
        config.productKey = deviceInfo.productKey;
        config.deviceName = deviceInfo.deviceName;
        config.deviceSecret = deviceInfo.deviceSecret;
        config.channelHost = url;
        /**
         * Specify whether to receive offline messages.
         * The cleanSession field that corresponds to the MQTT connection.
         */
        config.receiveOfflineMsg = false;
        params.mqttClientConfig = config;
        ALog.setLevel(LEVEL_DEBUG);
        ALog.i(TAG, "mqtt connetcion info=" + params);
        /**
         * Configure the initialization parameters and specify the certificate informatio
n about the device.
         */
        params.deviceInfo = deviceInfo;
        /** Establish a connection. **/
        LinkKit.getInstance().init(params, new ILinkKitConnectListener() {
            public void onError(AError aError) {
                ALog.e(TAG, "Init Error error=" + aError);
            }
            public void onInitDone(InitResult initResult) {
                ALog.i(TAG, "onInitDone result=" + initResult);
                List<Property> properties =   LinkKit.getInstance().getDeviceThing().getP
roperties();
                ALog.i(TAG, "The device properties" + JSON.toJSONString(properties));
                List<Event> getEvents  =   LinkKit.getInstance().getDeviceThing().getEven
ts();
                ALog.i(TAG, "List of device events" + JSON.toJSONString(getEvents));
                /* Submit properties. TODO: Make sure that the properties that you want t
o submit are defined in the TSL model of the product. Example: MicSwitch. Otherwise, an e
rror message is returned. */
                handlePropertySet("MicSwitch", new ValueWrapper.IntValueWrapper(1));
                /* Submit events. TODO: Make sure that the events that you want to submit
are defined in the TSL model of the product. Example: Offline_alarm. Otherwise, an error
message is returned. */
                Map<String,ValueWrapper> values = new HashMap<>();
                values.put("eventValue",new ValueWrapper.IntValueWrapper(0));
                OutputParams outputParams = new OutputParams(values);
                handleEventSet("Offline_alarm",outputParams);
            }
        });
    }
```

> **Note** The property and event identifiers in the code must be the same as the identifiers that are defined in the TSL model.

- Configure Link SDK to submit properties.

```
/**
     * The device submits properties in the Alink JSON format.
     * @param identifier: the identifier of the property.
     * @param value: the value of the property.
     * @return
     */
    private void handlePropertySet(String identifier, ValueWrapper value ) {
        ALog.i(TAG, "Identifier of the property=" + identifier);
        Map<String, ValueWrapper> reportData = new HashMap<>();
        reportData.put(identifier, value);
        LinkKit.getInstance().getDeviceThing().thingPropertyPost(reportData, new IPublish
ResourceListener() {
            public void onSuccess(String s, Object o) {
                // The property is submitted.
                ALog.i(TAG, "Submitted. onSuccess() called with: s = [" + s + "], o = ["
+ o + "]");
            }
            public void onError(String s, AError aError) {
                // The property value failed to be submitted.
                ALog.i(TAG, "Failed to submit. onError() called with: s = [" + s + "], aE
rror = [" + JSON.toJSONString(aError) + "]");
            }
        });
    }
```

- Configure Link SDK to submit events.

```
/**
     * The device submits events in the Alink JSON format.
     * @param identifier: the identifier of the event.
     * @param params: the parameters of the event.
     * @return
     */
    private void handleEventSet(String identifyID, OutputParams params ) {
        ALog.i(TAG, "Identifier of the event=" + identifyID + "  params=" + JSON.toJSONSt
ring(params));
        LinkKit.getInstance().getDeviceThing().thingEventPost( identifyID,  params, new I
PublishResourceListener() {
            public void onSuccess(String s, Object o) {
                // The event is submitted.
                ALog.i(TAG, "Submitted. onSuccess() called with: s = [" + s + "], o = ["
+ o + "]");
            }
            public void onError(String s, AError aError) {
                // Failed to submit the event.
                ALog.i(TAG, "Failed to submit. onError() called with: s = [" + s + "], aE
rror = [" + JSON.toJSONString(aError) + "]");
            }
        });
    }
```

Send commands to configure properties and call services by using an IoT Platform SDK

- Initialize a client.

  The *IotClient* file in the *java/src/main/com.aliyun.iot.client* directory contains the sample code.

```
public class IotClient {
  private static String accessKeyID;
  private static String accessKeySecret;
  private static String regionId;
  private static String domain;
  private static String version;
  public static DefaultAcsClient getClient() {
    DefaultAcsClient client = null;
    Properties prop = new Properties();
    try {
      prop.load(Object.class.getResourceAsStream("/config.properties"));
      accessKeyID = prop.getProperty("user.accessKeyID");
      accessKeySecret = prop.getProperty("user.accessKeySecret");
      regionId = prop.getProperty("iot.regionId");
            domain = prop.getProperty("iot.domain");
            version = prop.getProperty("iot.version");
      IClientProfile profile = DefaultProfile.getProfile(regionId, accessKeyID, accessKey
Secret);
      DefaultProfile.addEndpoint(regionId, regionId, prop.getProperty("iot.productCode"),
          prop.getProperty("iot.domain"));
      // Initialize the client.
      client = new DefaultAcsClient(profile);
    } catch (Exception e) {
      LogUtil.print("Failed to initialize the client. exception:" + e.getMessage());
    }
    return client;
  }
    public static String getRegionId() {
        return regionId;
    }
    public static void setRegionId(String regionId) {
        IotClient.regionId = regionId;
    }
    public static String getDomain() {
        return domain;
    }
    public static void setDomain(String domain) {
        IotClient.domain = domain;
    }
    public static String getVersion() {
        return version;
    }
    public static void setVersion(String version) {
        IotClient.version = version;
    }
}
```

- Initialize the CommonRequest public class.

  The *AbstractManager* file in the *java/src/main/com.aliyun.iot.api.common.openApi* directory encapsulates the CommonRequest public class.

```
public class AbstractManager {
    private static DefaultAcsClient client;
    static {
        client = IotClient.getClient();
    }
    /**
     *  Initialize the CommonRequest class. action: the name of the API operation.
     * domain: the endpoint.
     * version: the API version.
     */
    public static CommonRequest executeTests(String action) {
        CommonRequest request = new CommonRequest();
        request.setDomain(IotClient.getDomain());
        request.setMethod(MethodType.POST);
        request.setVersion(IotClient.getVersion());
        request.setAction(action);
        return request;
    }
```

- Call the API operations of IoT Platform to configure properties and call services.

  The *ThingManagerForPopSDk* file in the *java/src/main/com.aliyun.iot.api.common.openApi* directory contains the sample code.

  - Call the SetDeviceProperty operation to configure a device property.

```
public static void SetDeviceProperty(String InstanceId, String IotId, String ProductKey
, String DeviceName , String Items) {
        SetDevicePropertyResponse response =null;
        SetDevicePropertyRequest request=new SetDevicePropertyRequest();
        request.setDeviceName(DeviceName);
        request.setIotId(IotId);
        request.setItems(Items);
        request.setProductKey(ProductKey);
        request.setIotInstanceId(InstanceId);
        try {
            response = client.getAcsResponse(request);
            if (response.getSuccess() != null && response.getSuccess()) {
                LogUtil.print("The device property is configured.");
                LogUtil.print(JSON.toJSONString(response));
            } else {
                LogUtil.print("Failed to configure the device property.");
                LogUtil.error(JSON.toJSONString(response));
            }
        } catch (ClientException e) {
            e.printStackTrace();
            LogUtil.error("Failed to configure the device property." + JSON.toJSONStrin
g(response));
        }
    }
```

- Call the SetDevicesProperty operation to batch configure device properties.

```
/**
     * Batch configure device properties.
     *
     * @param ProductKey: the ProductKey of the product to which the devices whose prop
erties you want to configure belong.
     * @param DeviceNames: the DeviceNames of the devices whose properties you want to
configure.
     * @param Items: the property information that consists of multiple pairs of key-va
lue JSON strings. This parameter is required.
     *
     * @Des: the description.
     */
    public static void SetDevicesProperty(String InstanceId, String ProductKey, List<St
ring> DeviceNames, String Items) {
        SetDevicesPropertyResponse response = new SetDevicesPropertyResponse();
        SetDevicesPropertyRequest request = new SetDevicesPropertyRequest();
        request.setDeviceNames(DeviceNames);
        request.setItems(Items);
        request.setProductKey(ProductKey);
        request.setIotInstanceId(InstanceId);
        try {
            response = client.getAcsResponse(request);
            if (response.getSuccess() != null && response.getSuccess()) {
                LogUtil.print("The properties are batch configured.");
                LogUtil.print(JSON.toJSONString(response));
            } else {
                LogUtil.print("Failed to batch configure the device properties.");
                LogUtil.error(JSON.toJSONString(response));
            }
        } catch (ClientException e) {
            e.printStackTrace();
            LogUtil.error("Failed to batch configure the device properties." + JSON.toJ
SONString(response));
        }
    }
```

- Call the InvokeThingService operation to call a device service.

```
    /**
     * @param Identifier: the identifier of the service. This parameter is required.
     * @param Args: the input parameters that are required to start the service. This p
arameter is required.
     */
    public static InvokeThingServiceResponse.Data InvokeThingService(String InstanceId,
String IotId, String ProductKey, String DeviceName,
                                                                     String Identifier,
String Args) {
        InvokeThingServiceResponse response =null;
        InvokeThingServiceRequest request = new InvokeThingServiceRequest();
        request.setArgs(Args);
        request.setDeviceName(DeviceName);
        request.setIotId(IotId);
        request.setIdentifier(Identifier);
        request.setProductKey(ProductKey);
        request.setIotInstanceId(InstanceId);
        try {
            response = client.getAcsResponse(request);
            if (response.getSuccess() != null && response.getSuccess()) {
                LogUtil.print("The service is executed.");
                LogUtil.print(JSON.toJSONString(response));
            } else {
                LogUtil.print("Failed to execute the service.");
                LogUtil.error(JSON.toJSONString(response));
            }
            return response.getData();
        } catch (ClientException e) {
            e.printStackTrace();
            LogUtil.error("Failed to execute the service." + JSON.toJSONString(response
));
        }
        return null;
    }
```

Note If you want to synchronously call services, set the **Invoking Method** parameter to **Synchronization** when you define TSL models. When you develop devices, you must write the code to process synchronous service calls.

- Call the InvokeThingsService operation to batch call device services.

```
    /**
     * @param Identifier: the identifier of the service. This parameter is required.
     * @param Args: the input parameters that are required to start the service. This p
arameter is required.
     */
    public static void InvokeThingsService(String InstanceId, String IotId, String Prod
uctKey, List<String> DeviceNames,
                                           String Identifier, String Args) {
        InvokeThingsServiceResponse response =null;
        InvokeThingsServiceRequest request = new InvokeThingsServiceRequest();
        request.setArgs(Args);
        request.setIdentifier(Identifier);
        request.setDeviceNames(DeviceNames);
        request.setProductKey(ProductKey);
        request.setIotInstanceId(InstanceId);
        try {
            response = client.getAcsResponse(request);
            if (response.getSuccess() != null && response.getSuccess()) {
                LogUtil.print("The services are called.");
                LogUtil.print(JSON.toJSONString(response));
            } else {
                LogUtil.print("Failed to batch call the services.");
                LogUtil.error(JSON.toJSONString(response));
            }
        } catch (ClientException e) {
            e.printStackTrace();
            LogUtil.error("Failed to batch call the services." + JSON.toJSONString(resp
onse));
        }
    }
```

Sample request:

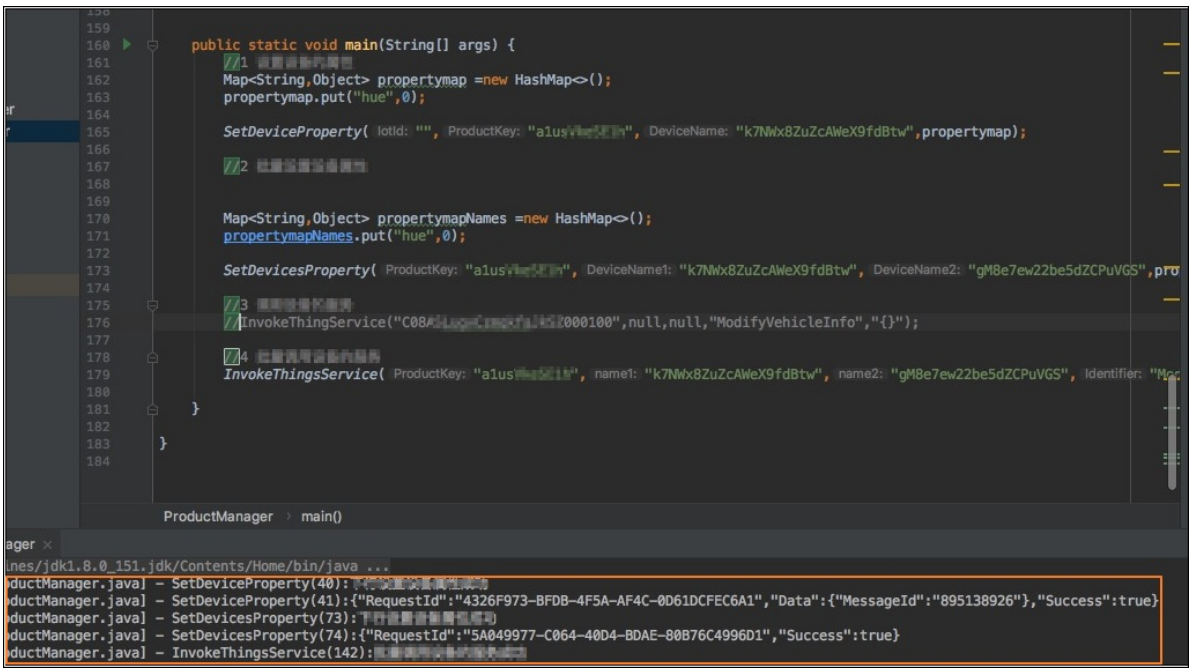
```
public static void main(String[] args) {
        /** The DeviceName of the online device and the Productkey of the product to which
the online device belongs. */
        String deviceName = "2pxuAQB2I7wGPmqq***";
        String deviceProductkey = "a1QbjI2***";
        /** If you use an Enterprise Edition instance or a public instance of the new versi
on, specify the instance ID for the InstanceId parameter. To obtain the instance ID, log on
to the IoT Platform console and view the instance ID on the Overview page.
         * If you use a public instance of the old version, leave the InstanceId parameter
empty"".
         */
        String InstanceId = "iot-***tl02";
        //1. Configure a device property.
        SetDeviceProperty(InstanceId, null, deviceProductkey, deviceName,"{\"hue\":0}");
        //2. Batch configure device properties.
        List<String>  deviceNames = new ArrayList<>();
        deviceNames.add(deviceName);
        SetDevicesProperty(InstanceId, deviceProductkey, deviceNames, "{\"hue\":0}");
        //3. Call a device service.
        InvokeThingService(InstanceId, null, deviceProductkey, deviceName, "ModifyVehicleIn
fo", "{}");
        //4. Batch call device services.
        List<String>  deviceNamesService = new ArrayList<>();
        deviceNamesService.add(deviceName);
        InvokeThingsService(null, deviceProductkey, deviceNamesService, "ModifyVehicleInfo"
, "{}");
    }
```

Debug the SDKs

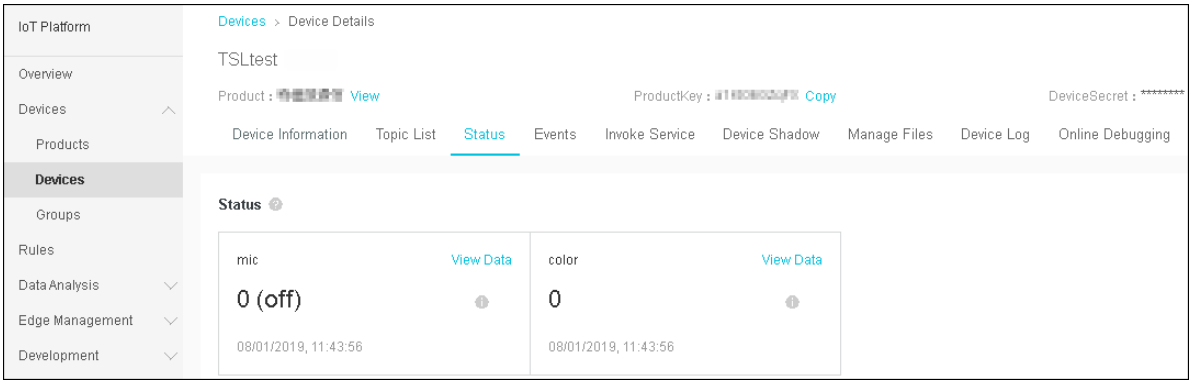
After you configure the Link SDK and the IoT Platform SDK, run the SDKs.

Check results:

- View local logs.



- On the **Device Details** page of the device, click **Default Module**.
  - The **Status** tab displays the property values that the device most recently submitted.
  - The **Events** tab displays the events that the device most recently submitted.
  - The **Invoke Service** tab displays the records of service calls.



2.4. Communication between devices

2.4.1. M2M communication

The machine-to-Machine (M2M) technology is used to achieve end-to-end communication. This article describes how to build an M2M communication architecture in IoT Platform by using the data forwarding feature of the rules engine or the topic-based message routing service. A connection between a smart lamp and a mobile app is used as an example.

IoT Platform handles communication requests between the device and the app. This allows you to focus on your business system. You do not need to worry about technical issues such as guaranteeing stable communication in high-concurrency scenarios or maintaining a large number of servers to handle requests.

For more information, see the following topics:

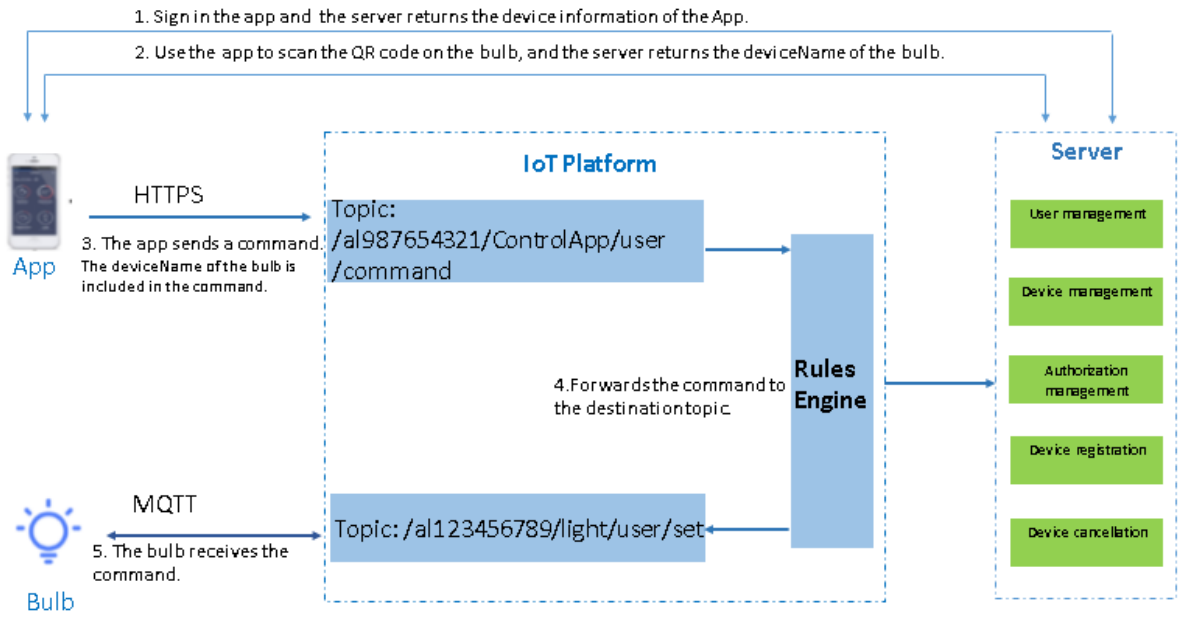
- Use the rules engine to establish M2M communication
- Use topic-based message routing to establish M2M communication

2.4.2. Use the rules engine to establish M2M communication

This article describes how to use the data forwarding feature of the rules engine in IoT Platform to build an machine-to-machine (M2M) communication architecture. A connection between a smart lamp and a mobile app is used as an example.

Context

The following figure shows how to control the smart lamp by using the mobile app.



Procedure

1. In the IoT Platform console, create a product and device for the smart lamp and define the features of the product. For more information, see Create a product, Create multiple devices at a time, and Add a TSL feature.

   In this example, the **ProductKey** and **DeviceName** parameters of the smart lamp are respectively set to al123456789 and light.

2. Develop a device SDK for the smart lamp.

   In this example, the device and IoT Platform use MQTT to communicate with each other.

   For information about how to develop a device SDK, see the Link SDK documentation.

3. In the IoT Platform console, create a product and a device for the mobile app.

   In this example, the **ProductKey** and **DeviceName** parameters of the mobile app are respectively set to al987654321 and ControlApp.

   When a registered user logs on to the mobile app, your server sends the device information to the mobile app. This way, the mobile app can connect to IoT Platform as a device.

4. Develop the mobile app.

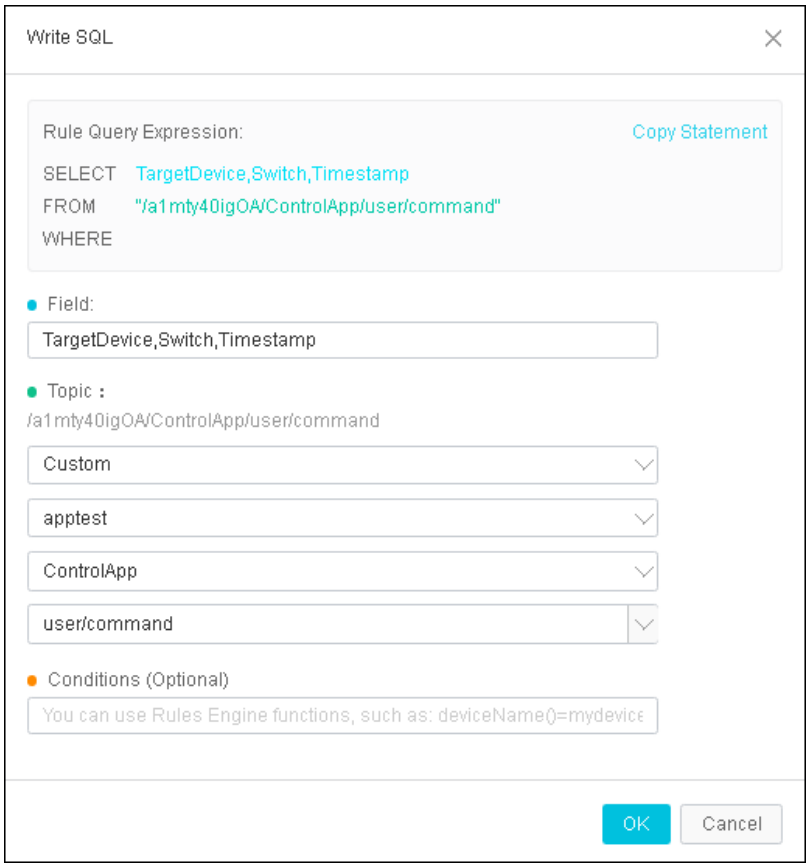
   In this example, the mobile app and IoT Platform use HTTPS to communicate with each other.

   Format of a command that the mobile app sends to the smart lamp:

```
{
    "TargetDevice": "light",
    "Switch": "off",
    "Timestamp": 1557750407000
}
```

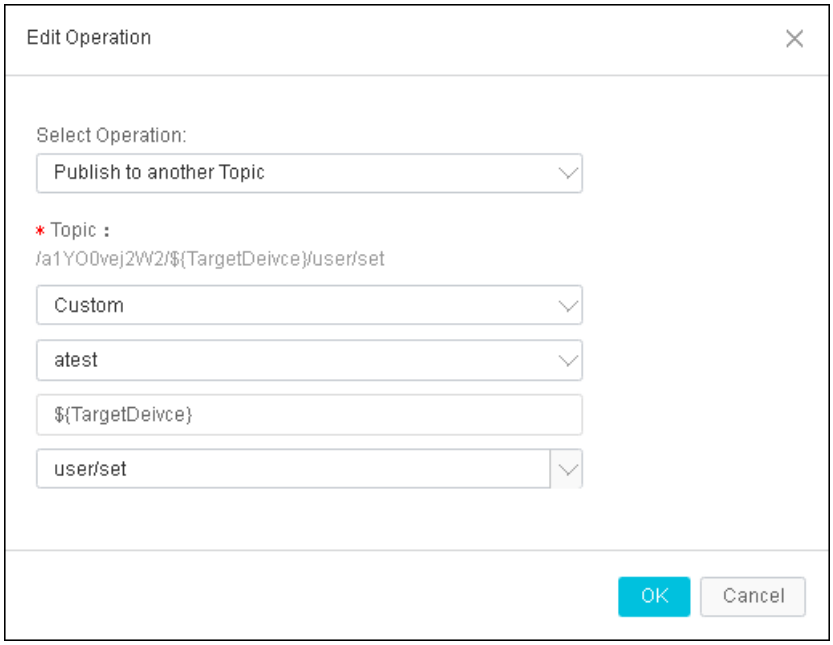
5. Develop the device SDK to connect the smart lamp with IoT Platform, receive the command, and implement the command.
6. Configure a data forwarding rule to forward the command from the mobile app to the topic of the smart lamp.
   i. Log on to the IoT Platform console. Choose **Rules > Data Forwarding**.
   ii. Click **Create Rule**. In the dialog box that appears, enter a rule name and click **OK**.
   iii. In the **A flow rule has been created** message, click **Edit**.
   iv. On the **Data Forwarding Rule** page, click **Write SQL** in the **Data Processing** section.
   v. In the **Write SQL** dialog box, write an SQL statement that is used to process the forwarded message, and then click **OK**. You can use the SQL statement to extract required fields from the message that is sent by the mobile app. Only the fields are sent to the smart lamp.

      In this example, the TargetDevice, Timestamp, and Switch fields are extracted.



vi. On the **Data Forwarding Rule** page, click **Add Operation** in the **Data Forwarding** section. In the dialog box that appears, set a forwarding message destination. In this example, specify a smart lamp topic that has the Subscribe permission as the destination.

> **Note**
> - Select **Publish to another Topic**.
> - When you specify devices, use `${TargetDevice}` to indicate all smart lamps. In this example, `${TargetDevice}` indicates the smart lamp named `light`.



7. The mobile app user scans the QR code to bind the app with the smart lamp.

   After the app sends a request to the server to bind a device, the server binds the smart lamp and returns the device name. The returned name is specified by the deviceName parameter. In this example, the device name is light.

8. The mobile app user sends a command from the app.
   i. The app sends the command to the topic in IoT Platform. In this example, the topic is `/al987654321/ControlApp/user/update`.
   ii. Then, IoT Platform sends the command to the topic of the smart lamp based on the data forwarding rule. In this example, the topic is `/al123456789/light/user/set`.
   iii. The smart lamp device receives the commands and performs the required operations.

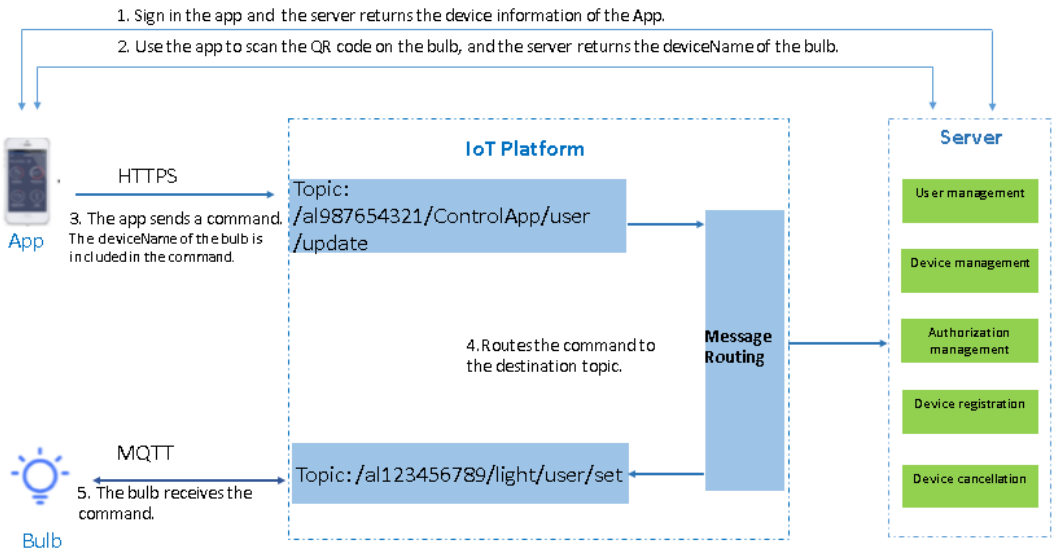
> **Note** The app can also send a request to your server to unbind the smart lamp device. After the device is unbound, you cannot use the app to control the smart lamp.

2.4.3. Use topic-based message routing to establish M2M communication

This article describes how to use the topic-based message routing service of IoT Platform to build an M2M communication architecture. A connection between a smart lamp and a mobile app is used as an example.

Context

The following figure shows how to control the smart lamp by using the mobile app.



Procedure

1. In the IoT Platform console, create a product and device for the smart lamp and define the features of the product. For more information, see Create a product, Create multiple devices at a time, and Add a TSL feature.

   In this example, the **ProductKey** and **DeviceName** parameters of the smart lamp are respectively set to al123456789 and light.

2. Develop a device SDK for the smart lamp.

   In this example, the device and IoT Platform use MQTT to communicate with each other.

   For information about how to develop a device SDK, see the Link SDK documentation.

3. In the IoT Platform console, create a product and a device for the mobile app.

   In this example, the ProductKey and DeviceName parameters of the mobile app are respectively set to al987654321 and ControlApp.

   When a registered user logs on to the mobile app, your server sends the device information to the mobile app. This way, the mobile app can connect to IoT Platform as a device.

4. Call the CreateTopicRouteTable operation to create a message routing relationship between the topics of the app and the smart lamp.
   - Set the SrcTopic parameter to the topic of the app: `/al987654321/ControlApp/user/update`.
   - Set the DstTopics parameter to the topic of the smart lamp: `/al123456789/light/user/set`.

5. Develop the mobile app.

   In this example, the mobile app and IoT Platform use HTTPS to communicate with each other.

   Format of a command that the mobile app sends to the smart lamp:

```
{
    "TargetDevice": "light",
    "Switch": "off",
    "Timestamp": 1557750407000
}
```

6. The mobile app user scans the QR code to bind the app with the smart lamp.

   After the app sends a request to the server to bind a device, the server binds the smart lamp and returns the device name. The returned name is specified by the deviceName parameter. In this example, the device name is light.

7. The mobile app user sends control commands from the app.

   i. The topic to which the app sends commands is `/a1987654321/ControlApp/user/update`. Only JSON format commands are supported.

   ii. IoT Platform routes commands to the topic of the smart lamp device based on the defined message routing relationship. The topic is `/a1123456789/light/user/set`.

   iii. The smart lamp device receives the commands and performs the required operations.

> **Note** You can configure the mobile app to send an unbinding request to the server. This way, the server calls the DeleteTopicRouteTable operation of IoT Platform to delete the message routing relationship. After the routing relationship is deleted, you cannot use the app to control the smart lamp.

2.5. Create an MNS server-side subscription

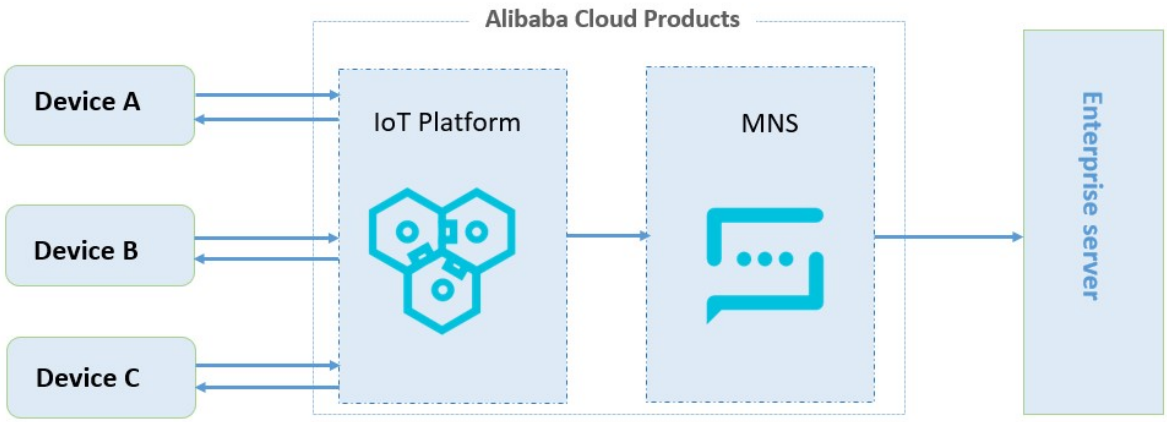
This article describes how to create a Message Service (MNS) server-side subscription to push status changes of the devices under a product to an MNS queue. Your server can receive the messages by listening to the MNS queue.

Prerequisites

- The following Alibaba Cloud services are activated:
  - IoT Platform
  - MNS
- The Java development environment Eclipse is installed.

Context

The following figure shows data flows.



Configure a server-side subscription

In the IoT Platform console, create an MNS server-side subscription and select the types of messages to which you want to subscribe.

1. Log on to the IoT Platform console.
2.
3. In the left-side navigation pane, choose **Devices > Products**. On the Products page, click **Create Product** to create a product.
4. In the left-side navigation pane, choose **Devices > Devices**. On the Devices page, click **Add Device** to create a device under the product.

   When you use Link SDK to develop a device, you must obtain the device certificate.
5. In the left-side navigation pane, choose **Rules Engine > Server-side Subscription**. On the Server-side Subscription page, click **Create Subscription** to create an MNS server-side subscription. For more information, see Configure MNS server-side subscriptions.

   > **Note** The first time you select MNS in the Subscription Type field, a message prompts you to authorize the access of IoT Platform to MNS. Click **Authorize Now** to go to the Resource Access Management (RAM) console. Click Confirm Authorization Policy.

   Set the Message Type parameter to **Device Status Change Notification**. All messages about status changes of the devices under the product are pushed to an MNS queue.

   After you create the subscription, IoT Platform automatically creates a queue in MNS. The queue is used to receive messages from IoT Platform. The format of the queue name is `aliyun-iot-${yourProductKey}`. When you use an MNS SDK to listen to the messages of a queue, you must specify the name of the queue.

   On the Server-side Subscription page, move the pointer over the icon next to MNS to view the name of a queue.

Use MNS SDK to receive messages

In this example, MNS SDK for Java is used.

1. To download the SDK, see Release notes of the SDK for Java. Download the *aliyun-sdk-mns-samples1.1.8* package, and then decompress the package.
2. Open Eclipse, click Import Project, and then select the *aliyun-sdk-mns-samples* directory.
3. In the *C:\Users\${YourComputerUserName}* local directory, create the *.aliyun-mns* .properties file. Add identity information in the following format to the file. The identity information is used by MNS for authentication.

```
mns.accountendpoint=http://$your_accountId.mns.$your_regionId.aliyuncs.com
mns.accesskeyid=$your_accesskeyid
mns.accesskeysecret=$your_accesskeysecret
```

Parameter	Description
accountendpoint	The endpoint of MNS. In the MNS console, select the region where the queue resides and click **Get Endpoint** to view the information.
accesskeyid	The AccessKey ID of your Alibaba Cloud account. To create or view your AccessKey pair, log on to the Alibaba Cloud Management Console. Move the pointer over your profile picture, and click **AccessKey** to go to the **User Management** page. On the page, you can create or view your AccessKey pairs.
accesskeysecret	The AccessKey secret of your Alibaba Cloud account. Obtain the AccessKey secret in the same way as the AccessKey ID.

4. Add the following code to the *ComsumerDemo* file in the *src\main\java\com.aliyun.mns.sample.Queue* directory. The code is used to specify the name of the queue that is automatically created by IoT Platform.

```
    public static void main(String[] args) {
        CloudAccount account = new CloudAccount(
                ServiceSettings.getMNSAccessKeyId(),
                ServiceSettings.getMNSAccessKeySecret(),
                ServiceSettings.getMNSAccountEndpoint());
        MNSClient client = account.getMNSClient(); // Initialize the client.
        // Retrieve messages.
        try{
            CloudQueue queue = client.getQueueRef("aliyun-iot-a1eN7La****");// Replace
this queue with the queue that is automatically created by IoT Platform.
            for (int i = 0; i < 10; i++)
            {
                Message popMsg = queue.popMessage(); // The long polling period.
                if (popMsg != null){
                    System.out.println("message handle:
                " + popMsg.getReceiptHandle());
                    System.out.println("message body:
                " + popMsg.getMessageBodyAsString()); // Retrieve the message payloads.
                    System.out.println("message id:
                " + popMsg.getMessageId());
                    System.out.println("message dequeue count:"
        + popMsg.getDequeueCount());                    // <<to add your special logic.
>>
                    // Delete messages from the queue.
                    queue.deleteMessage(popMsg.getReceiptHandle());
                    System.out.println("delete message successfully.\n");
                }
            }
        }
```

5. Run the ComsumerDemo file.

Configure a device SDK

1. To download a device SDK, see Download device SDKs. In this example, use the SDK for Java.
2. At the bottom of the Configure a project article, click Java SDK Demo to download the package, and then decompress the package.
3. In the Java development tool, import the *JavaLinkKit Demo* directory as a project.
4. In the *device_id* file, specify the device certificate information.

5. In the *MqttSample* file of the *src\devicesdk\demo* directory, add the information about the device, and specify the name of the topic to which device data is submitted.

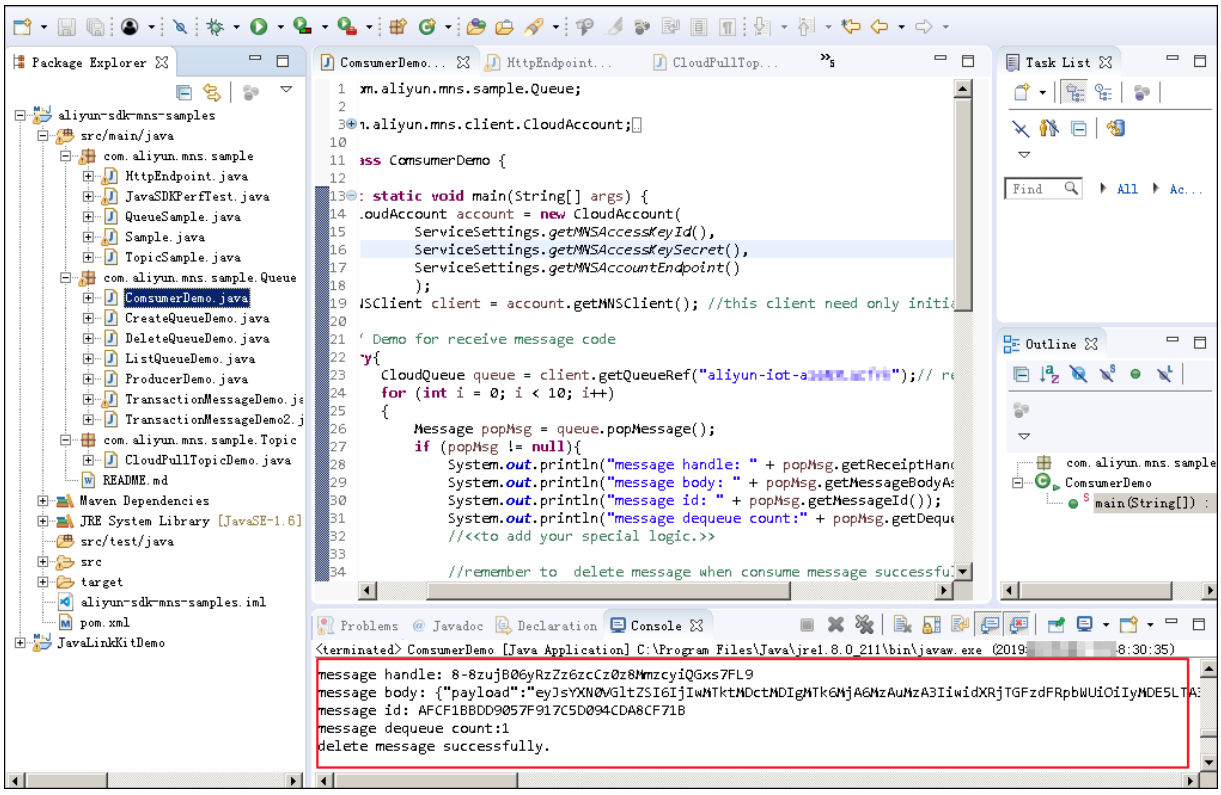
```
public class MqttSample extends BaseSample {
    final static String TAG = "MqttSample";

    public MqttSample(String a[illegible], String a[illegible]) {
        super(a[illegible]);
    }

    /**
     *
     */
    public void publish() {
        MqttPublishRequest request = new MqttPublishRequest();
        // topic
        request.topic = "/sys/" + productKey + "/" + deviceName + "/thing/deviceinfo/update";
        /**
```

6. Run the MqttSample file to connect the device with IoT Platform.

Verify the result

After you run the code, a message indicating that the device is online is sent to the MNS queue. You can use MNS SDK for Java to receive the message and then delete the message from the queue.

The following figure shows how to receive the message and then delete the message.

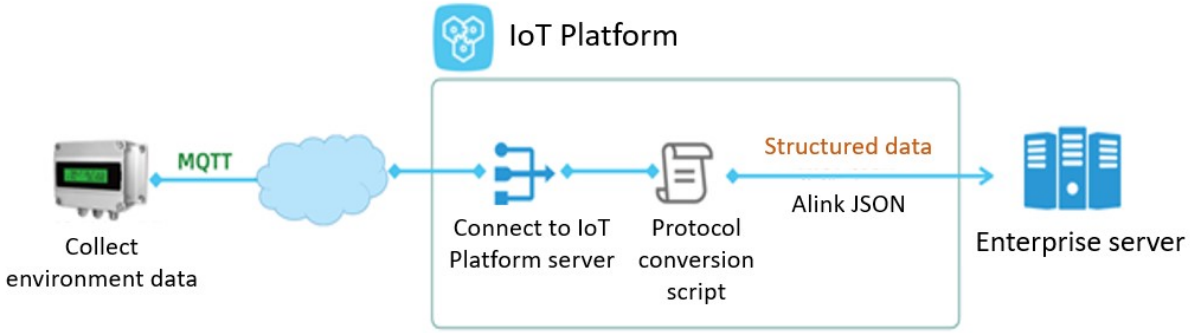


2.6. Parse pass-through data from devices

In some IoT scenarios, devices that have limited resources or low specifications cannot use Thing Specification Language (TSL) models in the JSON format to communicate with IoT Platform. To resolve the issue, you can pass through raw data from devices to IoT Platform. IoT Platform converts raw device data to Alink JSON data by using data parsing scripts that you submit. Then, you can perform subsequent operations on the converted data.

Context

The following figure shows how data flows between devices and enterprise servers.



In this example, a device that collects environmental data is used.

Connect a device to IoT Platform

1. Log on to the IoT Platform console.
2.
3. In the left-side navigation pane, choose **Devices > Products**. On the Products page, click **Create Product** to create a product.

   Set the **Data Type** parameter to **Custom**. For other parameters, use the default values.
4. After the product is created, click **Define TSL** to add TSL features. Then, publish the TSL model.

   This article provides a sample TSL model that you can import. For more information about how to import a TSL model, see Batch add TSL features.
5. In the left-side navigation pane, click **Devices**. Then, click **Add Device** to create a device under the product.

   After the device is added, obtain the certificate information of the device. The certificate information includes ProductKey, DeviceName, and DeviceSecret.
6. Develop a device and perform a test.

   Use Link SDK to simulate a device and submit Alink JSON data to IoT Platform. In this example, Link SDK for Node.js is used.

   For more information, see Link SDK.

   Sample code:

```
const mqtt = require('aliyun-iot-mqtt');
// 1. Specify the device certificate information.
var options = {
    productKey:
"g4yf***",
    deviceName:
    "Esensor",
    deviceSecret: "e14f81***",
    host:
    "iot-***.mqtt.iothub.aliyuncs.com"
};
// 1. Establish a connection.
const client = mqtt.getAliyunIotMqttClient(options);
// 2.
    Listen to commands from IoT Platform.
client.subscribe(`/${options.productKey}/${options.deviceName}/user/get`)
client.on('message', function(topic, message) {
    console.log("topic " + topic)
    console.log("message " + message)
})
setInterval(function() {
    // 3. Submit environmental data.
    client.publish(`/sys/${options.productKey}/${options.deviceName}/thing/event/proper
ty/post`, getPostData(), { qos: 0 });
}, 5 * 1000);
function getPostData() {
    const payloadJson = {
        id:
    Date.now(),
        version: "1.0",
        params:
{
            PM25:
Math.floor((Math.random() * 1000)),
            temperature: Math.floor((Math.random() * 60) - 10),
            humidity:
Math.floor((Math.random() * 100)),
            co2:
Math.floor((Math.random() * 10000)),
            hcho: Math.floor((Math.random() * 5)),
            lightLux:
Math.floor((Math.random() * 10000))
        },
        method:
"thing.event.property.post"
    }
    console.log("payloadJson " + JSON.stringify(payloadJson))
    return JSON.stringify(payloadJson);
}
```

After the device is connected to IoT Platform, you can view the device status on the **Devices** page. The device status is **Online**.

Click **View** in the Actions column. Then, click **TSL Data**. In this example, the data type of the product is set to **Custom**. Therefore, the submitted TSL data is not displayed on the **Status** tab.

Choose **Maintenance > Device Log > Cloud run log**. You can select **Device-to-cloud Messages** to view the submitted TSL data. The data is displayed in the hexadecimal format.

In this example, the following data is displayed: `0xaa1fc800003710ff0005d76b15001c013400ad04ffff0400ffff18003000ff2e`.

Write a data parsing script

In the IoT Platform console, you can edit, submit, and debug data parsing scripts.

1. Log on to the IoT Platform console.In the left-side navigation pane, choose **Devices > Products**.
2. On the **Products** page, find the product that you want to configure and click **View** in the Actions column.
3. On the **Product Details** page, click the **Data Parsing** tab.
4. On the **Data Parsing** tab, write a data parsing script in the **Edit Script** section.

   Edit the script based on the communication protocol that is used by the device. The following table shows the message body structure of the device.

Byte	Description	Remarks
12	Low byte of a PM2.5 value	Returns a PM2.5 value. Valid values: 0 to 999 ug/m^3.
13	High byte of a PM2.5 value	
14	Low byte of a temperature value × 10	Returns a temperature value. Valid values: -10°C to 50°C.
15	High byte of a temperature value × 10	
16	Low byte of a humidity value	Returns a humidity value. Valid values: 0 to 99%.
17	High byte of a humidity value	
18	Low byte of a carbon dioxide value	Returns a carbon dioxide value. Valid values: 0 to 9999 mg/m^3.
19	High byte of a carbon dioxide value	
22	Low byte of a formaldehyde value × 100	Returns a formaldehyde value. Valid values: 0 to 9.99.
23	High byte of a formaldehyde value × 100	
28	Low byte of an illuminance value	Returns an illuminance value, in lux.
29	High byte of an illuminance value	

In this example, the device only submits environmental data. Therefore, you only need to configure the rawDataToProtocol() function to parse upstream data. You do not need to configure the protocolToRawData() function.

The following code shows a sample script:

```
var PROPERTY_REPORT_METHOD = 'thing.event.property.post';
// Parse upstream data in a custom format to Alink JSON data.
function rawDataToProtocol(bytes) {
    var uint8Array = new Uint8Array(bytes.length);
    for (var i = 0; i < bytes.length; i++) {
        uint8Array[i] = bytes[i] & 0xff;
    }
    var dataView = new DataView(uint8Array.buffer, 0);
    var jsonMap = new Object();
        // The method that is used to submit properties.
        jsonMap['method'] = PROPERTY_REPORT_METHOD;
        // The version number of the protocol. Set the value to 1.0.
        jsonMap['version'] = '1.0';
        // The ID of the request.
        jsonMap['id'] = new Date().getTime();
        var params = {};
        // The number 12 or 13 indicates a PM2.5 value.
        params['PM25'] = (dataView.getUint8(13)*256+dataView.getUint8(12));
        // The number 14 or 15 indicates a temperature value.
        params['temperature'] = (dataView.getUint8(15)*256+dataView.getUint8(14))/10;
        // The number 16 or 17 indicates a humidity value.
        params['humidity'] = (dataView.getUint8(17)*256+dataView.getUint8(16));
        // The number 18 or 19 indicates a carbon dioxide value.
        params['co2'] = (dataView.getUint8(19)*256+dataView.getUint8(18));
        // The number 22 or 23 indicates a formaldehyde value.
        params['hcho'] = (dataView.getUint8(23)*256+dataView.getUint8(22))/100;
        // The number 28 or 29 indicates an illuminance value.
        params['lightLux'] = (dataView.getUint8(29)*256+dataView.getUint8(28));
        jsonMap['params'] = params;
    return jsonMap;
}
// Parse downstream Alink JSON data to the data in a custom format.
function protocolToRawData(json) {
    var payloadArray = [1];// This device only submits data and cannot receive downstre
am commands.
    return payloadArray;
}
// Parse topic data in a custom format to Alink JSON data.
function transformPayload(topic, rawData) {
    var jsonObj = {}
    return jsonObj;
}
```

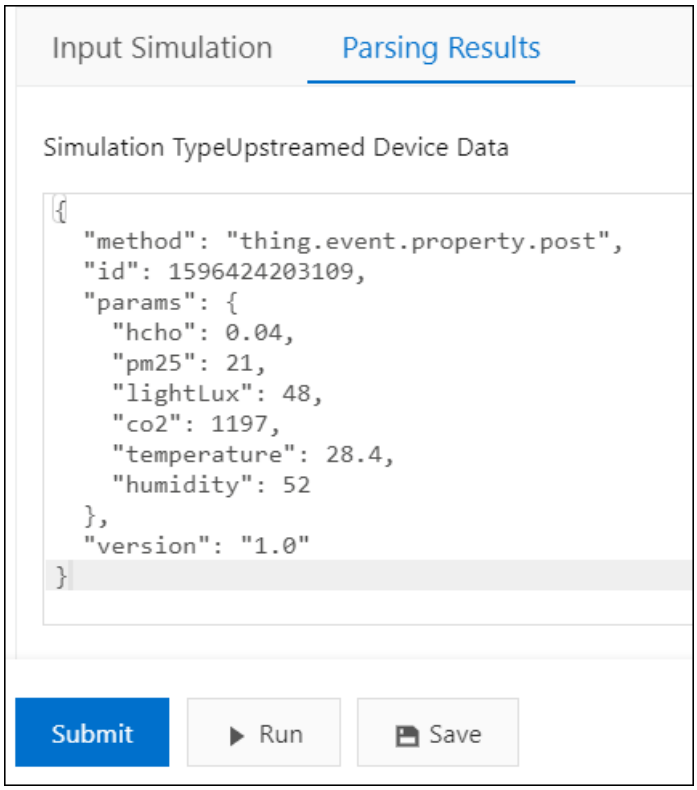
5. Test data parsing.
   i. Select **Upstreamed Device Data** in the Simulation Type field.

ii. On the **Input Simulation** tab, enter test data in the text box.

You can use the hexadecimal data that is submitted by the device as the test data. To view the hexadecimal data, go to the **Device Log** page. Example: `0xaa1fc800003710ff0005d76b15001c013400ad04ffff0400ffff18003000ff2e`.

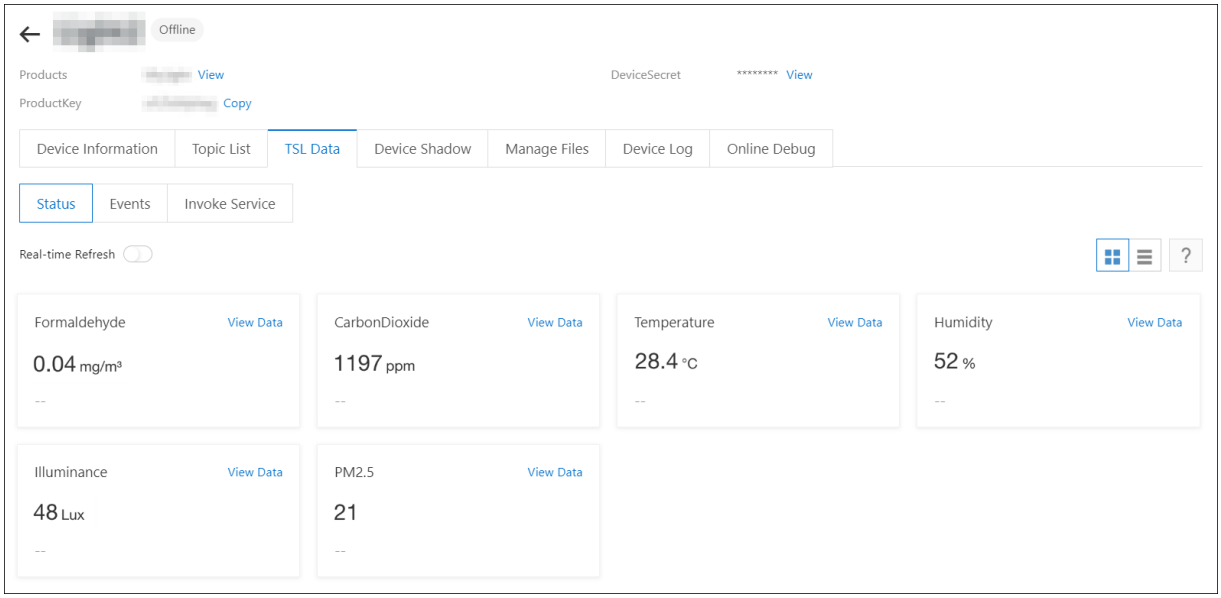
iii. Click **Run**.

The following figure shows the result on the **Parsing Results** tab.



6. After you confirm that test data can be parsed by the script, click **Submit** to submit the script to IoT Platform.

After you submit the script, you can use the device SDK to debug the script. You can submit data. Then, IoT Platform uses the script to parse the data. To view the parsed data, go to the **Device Details** page. Choose **TSL Data > Status**.



Example of upstream TSL data

```
{
  "schema": "https://iotx-tsl.oss-ap-southeast-1.aliyuncs.com/schema.json",
  "profile":
  {
    "version": "1.0",
    "productKey":
    "g4***"
  },
  "properties": [
    {
      "identifier":
    "lightLux",
      "name": "illuminance",
      "accessMode":
  "rw",
      "required": false,
      "dataType":
  {
        "type": "float",
        "specs":
    {
          "min":
      "0",
          "max": "10000",
          "unit":
      "Lux",
          "unitName": "luminous flux per unit area",
          "step":
      "0.1"
        }
      }
    },
    {
      "identifier": "PM25",
```

```
      "name":
      "PM25",
      "accessMode":
      "rw",
      "required": false,
      "dataType": {
        "type":
        "int",
        "specs": {
          "min":
        "0",
          "max": "1000",
          "unit":
          "μg/m³",
          "unitName": "micrograms per cubic meter",
          "step":
          "1"
        }
      }
    },
    {
      "identifier": "hcho",
      "name":
          "formaldehyde",
      "accessMode": "rw",
      "desc":
          "HCHOValue",
      "required": false,
      "dataType": {
        "type":
          "float",
        "specs": {
          "min":
        "0",
          "max":
      "10",
          "unit":
    "ppm",
          "unitName":
    "parts per million",
          "step":
      "0.1"
        }
      }
    },
    {
      "identifier": "co2",
      "name":
      "carbon dioxide",
      "accessMode": "rw",
      "required": false,
      "dataType":
      {
        "type": "int",
```

```
        "specs":
      {
          "min":
      "0",
          "max": "10000",
          "unit":
        "mg/m³",
          "unitName": "micrograms per cubic meter",
          "step":
        "1"
        }
      }
    },
    {
      "identifier": "humidity",
      "name":
          "humidity",
      "accessMode": "rw",
      "required": false,
      "dataType":
          {
        "type": "int",
        "specs":
          {
          "min": "0",
          "max":
          "100",
          "unit": "%",
          "unitName":
          "percentage",
          "step": "1"
        }
      }
    },
    {
      "identifier":
        "temperature",
      "name":
      "temperature",
      "accessMode":
    "rw",
      "desc":
    "motor working temperature",
      "required": false,
      "dataType":
      {
        "type": "float",
        "specs":
      {
          "min": "-10",
          "max":
      "50",
          "unit": "℃",
          "step":
```

```
      "0.1"
        }
      }
    }
  ],
  "events": [
    {
      "identifier":
      "post",
      "name":
      "post",
      "type": "info",
      "required": true,
      "desc":
        "submit properties",
      "method": "thing.event.property.post",
      "outputData":
        [
        {
          "identifier": "lightLux",
          "name":
          "illuminance",
          "dataType": {
            "type":
          "float",
            "specs": {
              "min":
          "0",
              "max": "10000",
              "unit":
          "Lux",
              "unitName": "luminous flux per unit area",
          "step":
          "0.1"
            }
          }
        },
        {
          "identifier": "PM25",
          "name":
        "PM25",
          "dataType":
    {
            "type":
  "int",
            "specs":
  {
              "min":
    "0",
              "max": "1000",
              "unit":
    "μg/m³",
              "unitName": micrograms per cubic meter",
          "step":
```

```
      "1"
            }
          }
        },
        {
          "identifier": "hcho",
          "name":
      "formaldehyde",
          "dataType":
      {
            "type": "float",
            "specs":
        {
              "min": "0",
              "max":
        "10",
              "unit": "ppm",
              "unitName":
          "parts per million",
          "step": "0.1"
            }
          }
        },
        {
          "identifier":
          "co2",
          "name": "carbon dioxide",
          "dataType":
          {
            "type": "int",
            "specs":
          {
              "min": "0",
              "max":
          "10000",
              "unit": "mg/m³",
              "unitName":
        "micrograms per cubic meter",
          "step":
      "1"
            }
          }
        },
        {
          "identifier":
    "humidity",
          "name":
    "humidity",
          "dataType":
        {
            "type": "int",
            "specs":
        {
              "min": "0",
              "max":
```

```
                max":
      "100",
              "unit": "%",
              "unitName":
      "percentage",
          "step":
      "1"
            }
          }
        },
        {
          "identifier": "temperature",
          "name":
        "temperature",
          "dataType": {
            "type":
        "float",
            "specs": {
              "min":
          "-10",
              "max": "50",
              "unit":
          "℃",
              "step": "0.1"
            }
          }
        }
      ]
    }
  ],
  "services":
          [
    {
      "identifier": "set",
      "name":
          "set",
      "required": true,
      "callType": "async",
      "desc":
          "set properties",
      "method": "thing.service.property.set",
      "inputData":
        [
        {
          "identifier":
      "lightLux",
          "name":
    "illuminance",
          "dataType":
    {
            "type":
      "float",
          "specs": {
            "min":
      "0",
```

```
            "max": "10000",
            "unit":
    "Lux",
            "unitName": "luminous flux per unit area",
        "step":
    "0.1"
          }
        }
      },
      {
        "identifier": "PM25",
        "name":
    "PM25",
        "dataType":
    {
          "type": "int",
          "specs":
      {
            "min": "0",
            "max":
      "1000",
            "unit": "μg/m³",
            "unitName":
        "micrograms per cubic meter",
        "step": "1"
          }
        }
      },
      {
        "identifier":
        "hcho",
        "name": "formaldehyde",
        "dataType":
        {
          "type": "float",
          "specs":
        {
            "min": "0",
            "max":
       "10",
            "unit":
    "ppm",
            "unitName":
  "parts per million",
        "step":
"0.1"
          }
        }
      },
      {
        "identifier":
"co2",
        "name": "carbon dioxide",
        "dataType":
```

```
{
        "type":
  "int",
        "specs": {
          "min":
  "0",
          "max": "10000",
          "unit":
  "mg/m³",
          "unitName": "micrograms per cubic meter",
      "step":
  "1"
        }
      }
    },
    {
      "identifier":
  "humidity",
      "name": "humidity",
      "dataType":
  {
        "type": "int",
        "specs":
  {
          "min": "0",
          "max":
    "100",
        "unit":
      "%",
        "unitName": "percentage",
      "step":
      "1"
        }
      }
    },
    {
      "identifier": "temperature",
      "name":
      "humidity",
      "dataType": {
        "type":
        "float",
        "specs": {
          "min":
        "-10",
          "max": "50",
          "unit":
          "℃",
          "step": "0.1"
        }
      }
    }
  ],
  "outputData":
```

```
            []
    },
    {
      "identifier": "get",
      "name":
              "get",
      "required": true,
      "callType": "async",
      "desc":
              "obtain properties",
      "method": "thing.service.property.get",
      "inputData":
              [
        "lightLux",
        "PM25",
        "hcho",
        "co2",
        "humidity",
        "temperature"
      ],
      "outputData": [
        {
          "identifier":
            "lightLux",
          "name":
          "illuminance",
          "dataType":
        {
            "type":
        "float",
            "specs":
          {
              "min": "0",
              "max":
          "10000",
              "unit": "Lux",
              "unitName":
          "luminous flux per unit area",
          "step": "0.1"
            }
          }
        },
        {
          "identifier":
            "PM25",
          "name": "PM25",
          "dataType":
            {
          "type": "int",
          "specs":
            {
            "min": "0",
            "max":
            "1000",
```

```
            "unit": "μg/m³",
            "unitName":
            "micrograms per cubic meter",
          "step": "1"
            }
          }
        },
        {
          "identifier":
              "hcho",
          "name": "formaldehyde",
          "dataType":
              {
            "type": "float",
            "specs":
            {
              "min":
          "0",
              "max":
        "10",
              "unit":
        "ppm",
              "unitName":
          "parts per million",
          "step": "0.1"
            }
          }
        },
        {
          "identifier":
          "co2",
          "name": "carbon dioxide",
          "dataType":
          {
            "type": "int",
            "specs":
            {
              "min": "0",
              "max":
            "10000",
              "unit": "mg/m³",
              "unitName":
              "micrograms per cubic meter",
          "step": "1"
            }
          }
        },
        {
          "identifier":
              "humidity",
          "name": "humidity",
          "dataType":
              {
            "type": "int",
            "specs":
```

```
            "specs":
              {
              "min": "0",
              "max":
              "100",
              "unit": "%",
              "unitName":
            "percentage",
          "step":
          "1"
            }
          }
        },
        {
          "identifier":
        "temperature",
          "name":
        "temperature",
          "dataType":
          {
            "type": "float",
            "specs":
          {
              "min": "-10",
              "max":
          "50",
              "unit": "℃",
              "step":
            "0.1"
            }
          }
        }
      ]
    }
  ]
}
```

3.Device management

3.1. Set a desired property value to control the bulb status

You can set desired property values in IoT Platform to control device properties. This article describes how to set a desired property value in IoT Platform to control the bulb status.

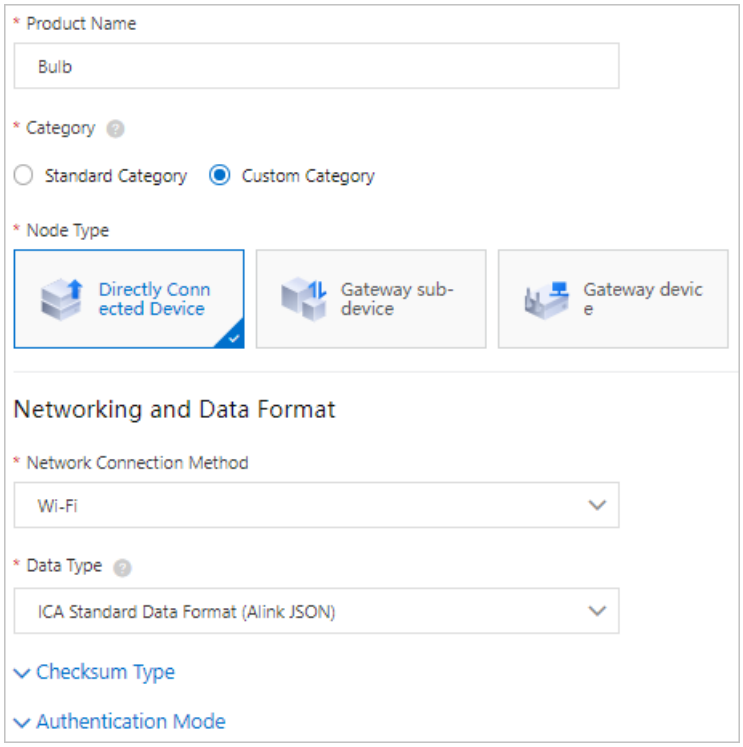
Context

After a bulb device is connected to IoT Platform, you must keep the bulb online all the time to control the bulb status. The bulb status in IoT Platform includes On (1) and Off (0). However, the bulb may not be online all the time in production environment.

You can set and store the desired property value of the bulb in IoT Platform. After the device goes online, it can read the desired property value that is stored in IoT Platform and update the property value. Then, the updated property value is submitted to IoT Platform and displayed in the IoT Platform device status.

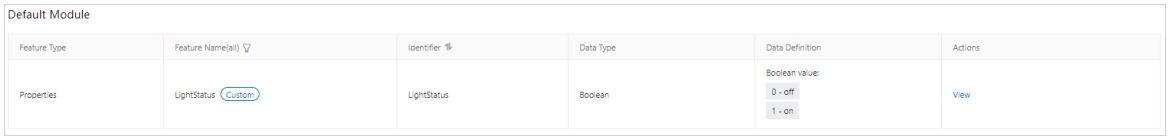
Create a product and a device

1. Log on to the IoT Platform console.
2. In the left-side navigation pane, choose **Devices > Products**. On the Products page, click **Create Product** to create a bulb product.

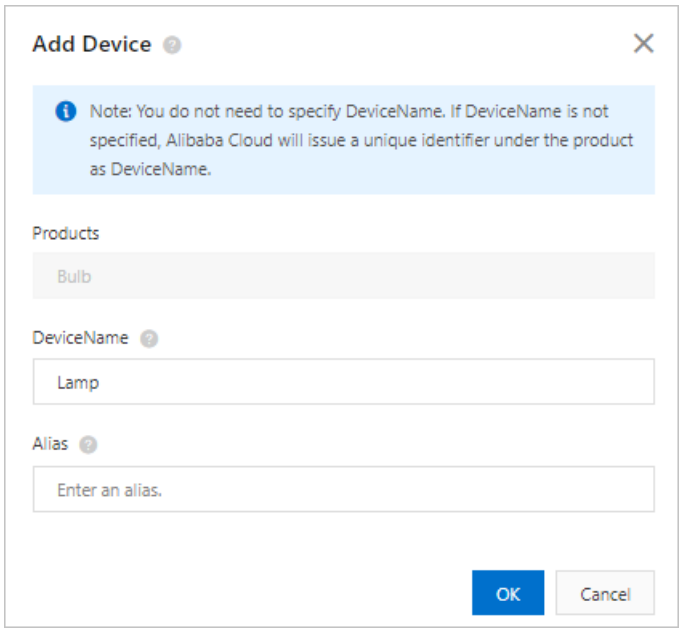


3. After you create the product, click **Create TSL** to add Thing Specification Language (TSL) features to the product and publish the TSL model. For more information, see Add a TSL feature.

   In this example, add the LightStatus property.

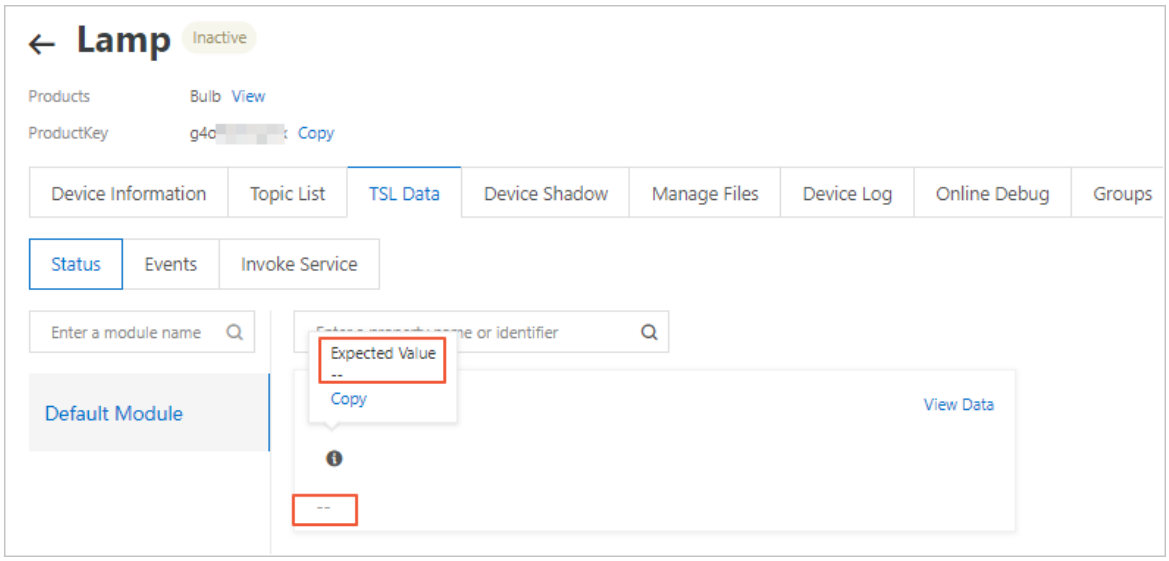


4. In the left-side navigation pane, choose **Devices > Devices**, and click **Add Device** to add the Lamp device to the product.



After you add the device, obtain the certificate information of the device. The certificate information includes the ProductKey, DeviceName, and DeviceSecret.

In the device list, click **View** to go to the **Device Details** page. You can view the device property value and desired property value on the **Status** tab. In this example, both the values are empty. The version of the desired property value is 0.



Set and query the desired property value in IoT Platform

You can set and query the latest desired property values of a device in IoT Platform by calling API operations.

For more information, see API operations. In this example, IoT Platform SDK for Java is used.

- Call the SetDeviceDesiredProperty operation to set desired property values of the device.

```
DefaultProfile profile = DefaultProfile.getProfile(
        "<RegionId>", // The region ID.
        "<accessKey>", // The AccessKey ID of the Alibaba Cloud account.
        "<accessSecret>"); // The AccessKey secret of Alibaba Cloud account.
IAcsClient client = new DefaultAcsClient(profile);
// Create an API request and set the parameters.
SetDeviceDesiredPropertyRequest request = new SetDeviceDesiredPropertyRequest();
request.setIotInstanceId("iot-060***");
request.setDeviceName("Lamp");
request.setProductKey("g4r***");
//The identifiers and desired values of the properties that you want to set.
request.setItems("{\"LightStatus\": 1}");
request.setVersions("{\"LightStatus\": 0}");
// Send the request and handle the response or exception.
try {
    SetDeviceDesiredPropertyResponse response = client.getAcsResponse(request);
    System.out.println(new Gson().toJson(response));
} catch (ServerException e) {
    e.printStackTrace();
} catch (ClientException e) {
    System.out.println("ErrCode:" + e.getErrCode());
    System.out.println("ErrMsg:" + e.getErrMsg());
    System.out.println("RequestId:" + e.getRequestId());
}
```

- Call the QueryDeviceDesiredProperty operation to query the desired property values of the device.

```
DefaultProfile profile = DefaultProfile.getProfile(
        "<RegionId>", // The region ID.
        "<accessKey>", // The AccessKey ID of the Alibaba Cloud account.
        "<accessSecret>"); // The AccessKey secret of Alibaba Cloud account.
IAcsClient client = new DefaultAcsClient(profile);
// Create an API request and set the parameters.
QueryDeviceDesiredPropertyRequest request = new QueryDeviceDesiredPropertyRequest();
request.setIotInstanceId("iot-060a02fq");
request.setProductKey("g4rmjb5q40O");
request.setDeviceName("Lamp");
// The identifiers of the properties that you want to query. If you do not specify a prop
erty identifier, the desired values of all properties except the read-only properties are
queried.
List<String> identifierList = new ArrayList<String>();
identifierList.add("LightStatus");
request.setIdentifiers(identifierList);
// Send the request and handle the response or exception.
try {
    QueryDeviceDesiredPropertyResponse response = client.getAcsResponse(request);
    System.out.println(new Gson().toJson(response));
} catch (ServerException e) {
    e.printStackTrace();
} catch (ClientException e) {
    System.out.println("ErrCode:" + e.getErrCode());
    System.out.println("ErrMsg:" + e.getErrMsg());
    System.out.println("RequestId:" + e.getRequestId());
}
```

For more information about how to set the parameters in the code, see Use IoT Platform SDK for Java.

After you set the desired property value of the Lamp device, the value is displayed on the Status tab in IoT Platform.

Develop the device

The bulb device can obtain desired property values in the following scenarios:

- If the bulb device goes from offline to online, it requests the desired property value that is cached in IoT Platform.
- If the bulb device is online, it receives the desired property value that is pushed by IoT Platform in real time.

For more information, see Download device SDKs.

This article provides complete device-side sample code. For more information, see Appendix: Sample code to configure a device.

1. Specify the certificate information and region ID of the device.

```
/**
* The certificate information about the device.
*/
private static String productKey = "******";
private static String deviceName = "********";
private static String deviceSecret = "***************";
/**
* The MQTT connection information.
*/
private static String regionId = "******";
```

Note

- For more information about the device certificate, see Create a product and a device.
- regionId specifies the ID of the region where your service resides. You can view the region in the upper-left corner of the IoT Platform console. For more information about region IDs, see Regions and zones.

2. Add the following methods. These methods can be used to change the actual property values of the bulb and automatically report the changes to IoT Platform.

```
/**
 * When the device handles a property value change, the methods are called in the follo
wing scenarios:
 * Scenario 1. Pull mode: After the device is connected to IoT Platform again, the devi
ce automatically requests the latest desired property value from IoT Platform.
 * Scenario 2. Push mode: When the device is online, it receives the desired property v
alues that IoT Platform pushes to the property.set topic.
 * @param identifier: the property identifier.
 * @param value: the desired property value.
 * @param needReport: specifies whether to report the property value to IoT Platform by
using the property.post topic.
 * In Scenario 2, the property report capability is integrated into the processing func
tion and the needReport parameter is set to false.
 * @return
 */
private boolean handlePropertySet(String identifier, ValueWrapper value, boolean needRe
port) {
    ALog.d(TAG, "The device handles property changes= [" + identifier + "], value = ["
+ value + "]");
    // Check whether the property settings are successful based on the response. In thi
s example, a success message is returned.
    boolean success = true;
    if (needReport) {
        reportProperty(identifier, value);
    }
    return success;
}
private void reportProperty(String identifier, ValueWrapper value){
    if (StringUtils.isEmptyString(identifier) || value == null) {
        return;
    }
    ALog.d(TAG, "Report property identity=" + identifier);
    Map<String, ValueWrapper> reportData = new HashMap<>();
    reportData.put(identifier, value);
    LinkKit.getInstance().getDeviceThing().thingPropertyPost(reportData, new IPublishRe
sourceListener() {
        public void onSuccess(String s, Object o) {
            // The property value is reported.
            ALog.d(TAG, "Report success onSuccess() called with: s = [" + s + "], o = [
" + o + "]");
        }
        public void onError(String s, AError aError) {
            // The property value fails to be reported.
            ALog.d(TAG, "Report failure onError() called with: s = [" + s + "], aError
= [" + JSON.toJSONString(aError) + "]");
        }
    });
}
```

3. If the bulb is online and a desired property value is set for the bulb in IoT Platform, IoT Platform pushes the value to the bulb. The bulb processes the message and changes the status.

   In the following code, the `connectNotifyListener()` method is called to process the message. For information about the Alink protocol, see Devices submit property information to IoT Platform.

After the device receives an asynchronous message from IoT Platform, the device calls the `mCommonHandler()` method and then the `handlePropertySet` method to update the property value.

```
/**
 * Register a function to respond to service calls and property settings.
 * When IoT Platform calls a service from the device, the device must respond to the call and return a response.
 */
public void connectNotifyListener() {
    List<Service> serviceList = LinkKit.getInstance().getDeviceThing().getServices();
    for (int i = 0; serviceList != null && i < serviceList.size(); i++) {
        Service service = serviceList.get(i);
        LinkKit.getInstance().getDeviceThing().setServiceHandler(service.getIdentifier(), mCommonHandler);
    }
}
private ITResRequestHandler mCommonHandler = new ITResRequestHandler() {
    public void onProcess(String serviceIdentifier, Object result, ITResResponseCallback itResResponseCallback) {
        ALog.d(TAG, "onProcess() called with: s = [" + serviceIdentifier + "]," +
                " o = [" + result + "], itResResponseCallback = [" + itResResponseCallback + "]");
        ALog.d(TAG, "Received an asynchronous service call from IoT Platform " + serviceIdentifier);
        try {
            if (SERVICE_SET.equals(serviceIdentifier)) {
                Map<String, ValueWrapper> data = (Map<String, ValueWrapper>)((InputParams)result).getData();
                ALog.d(TAG, "Received asynchronous downstream data " + data);
                //Set the property of the device and then report the property value to IoT Platform.
                boolean isSetPropertySuccess =
                        handlePropertySet("LightStatus", data.get("LightStatus"), false);
                if (isSetPropertySuccess) {
                    if (result instanceof InputParams) {
                        // Return a response to IoT Platform.
                        itResResponseCallback.onComplete(serviceIdentifier, null, null);
                    } else {
                        itResResponseCallback.onComplete(serviceIdentifier, null, null);
                    }
                } else {
                    AError error = new AError();
                    error.setCode(100);
                    error.setMsg("setPropertyFailed.");
                    itResResponseCallback.onComplete(serviceIdentifier, new ErrorInfo(error), null);
                }
            } else if (SERVICE_GET.equals(serviceIdentifier)) {
            } else {
                // The device operation varies by service.
                ALog.d(TAG, "Return a response to IoT Platform.");
```

```
                OutputParams outputParams = new OutputParams();
                // outputParams.put("op", new ValueWrapper.IntValueWrapper(20));
                itResResponseCallback.onComplete(serviceIdentifier, null, outputParams)
;
            }
        } catch (Exception e) {
            e.printStackTrace();
            ALog.d(TAG, "The format of the returned data is invalid");
        }
    }
    public void onSuccess(Object o, OutputParams outputParams) {
        ALog.d(TAG, "onSuccess() called with: o = [" + o + "], outputParams = [" + outp
utParams + "]");
        ALog.d(TAG, "The service was registered");
    }
    public void onFail(Object o, ErrorInfo errorInfo) {
        ALog.d(TAG, "onFail() called with: o = [" + o + "], errorInfo = [" + errorInfo
+ "]");
        ALog.d(TAG, "Service registration failed.");
    }
};
```

4. If the bulb is offline and a desired property value is set for the bulb in IoT Platform, the value is stored in IoT Platform.

   After the bulb goes online, it requests the desired property value from IoT Platform and call the `handlePropertySet()` method to update the property value.

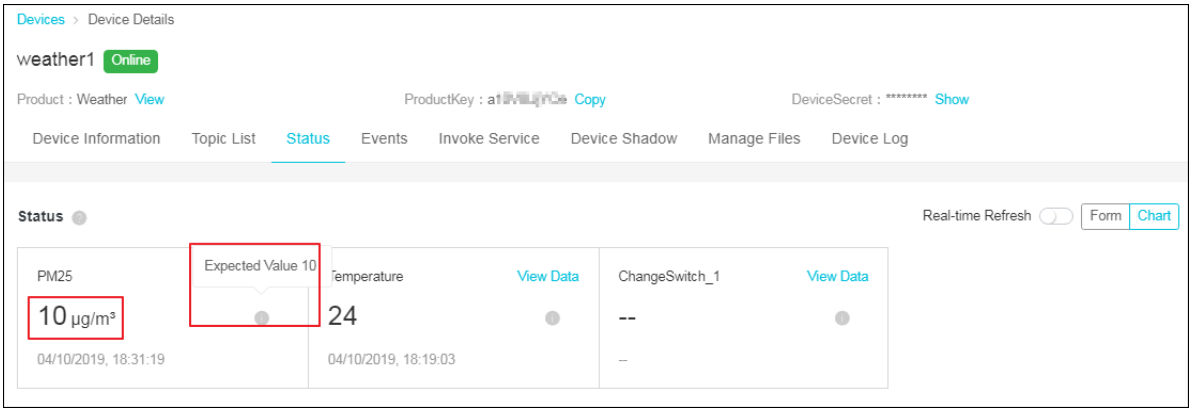
```
LinkKit.getInstance().init(params, new ILinkKitConnectListener() {
    public void onError(AError aError) {
        ALog.e(TAG, "Init Error error=" + aError);
    }
    public void onInitDone(InitResult initResult) {
        ALog.i(TAG, "onInitDone result=" + initResult);
        connectNotifyListener();
        // Request the latest desired property value from IoT Platform.
        getDesiredProperty(deviceInfo, Arrays.asList("LightStatus"), new IConnectSendLi
stener() {
            public void onResponse(ARequest aRequest, AResponse aResponse) {
                if(aRequest instanceof MqttPublishRequest && aResponse.data != null) {
                    JSONObject jsonObject = JSONObject.parseObject(aResponse.data.toStr
ing());
                    ALog.i(TAG, "onResponse result=" + jsonObject);
                    JSONObject dataObj = jsonObject.getJSONObject("data");
                    if (dataObj != null) {
                        if (dataObj.getJSONObject("LightStatus") == null) {
                            // No desired value is set.
                        } else {
                            Integer value = dataObj.getJSONObject("LightStatus").getInt
eger("value");
                            handlePropertySet("LightStatus", new ValueWrapper.IntValueW
rapper(value), true);
                        }
                    }
                }
```

```
            }
            public void onFailure(ARequest aRequest, AError aError) {
                ALog.d(TAG, "onFailure() called with: aRequest = [" + aRequest + "], aE
rror = [" + aError + "]");
            }
        });
    }
});
private void getDesiredProperty(BaseInfo info, List<String> properties, IConnectSendLis
tener listener) {
    ALog.d(TAG, "getDesiredProperty() called with: info = [" + info + "], listener = ["
+ listener + "]");
    if(info != null && !StringUtils.isEmptyString(info.productKey) && !StringUtils.isEm
ptyString(info.deviceName)) {
        MqttPublishRequest request = new MqttPublishRequest();
        request.topic = DESIRED_PROPERTY_GET.replace("{productKey}", info.productKey).r
eplace("{deviceName}", info.deviceName);
        request.replyTopic = DESIRED_PROPERTY_GET_REPLY.replace("{productKey}", info.pr
oductKey).replace("{deviceName}", info.deviceName);
        request.isRPC = true;
        RequestModel<List<String>> model = new RequestModel<>();
        model.id = String.valueOf(IDGeneraterUtils.getId());
        model.method = METHOD_GET_DESIRED_PROPERTY;
        model.params = properties;
        model.version = "1.0";
        request.payloadObj = model.toString();
        ALog.d(TAG, "getDesiredProperty: payloadObj=" + request.payloadObj);
        ConnectSDK.getInstance().send(request, listener);
    } else {
        ALog.w(TAG, "getDesiredProperty failed, baseInfo Empty.");
        if(listener != null) {
            AError error = new AError();
            error.setMsg("BaseInfoEmpty.");
            listener.onFailure(null, error);
        }
    }
}
```

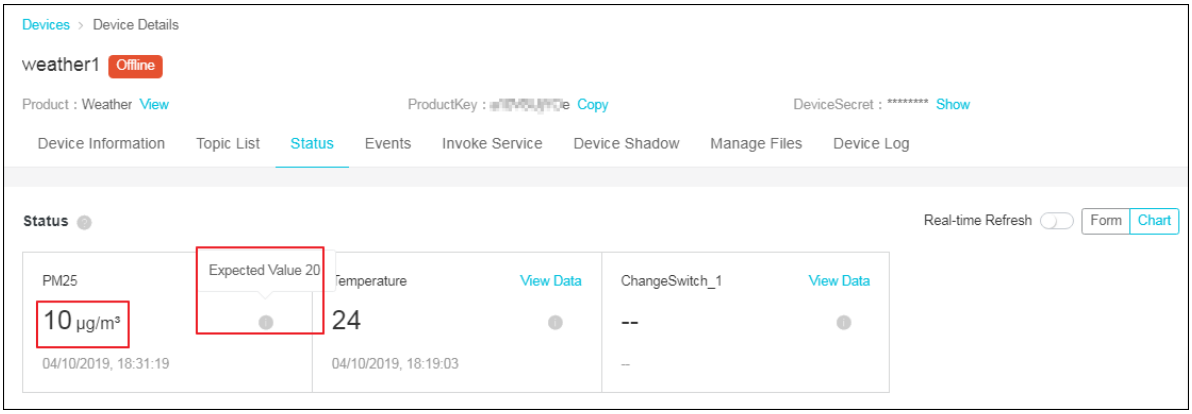
Verify the result

Run the sample code based on scenarios to verify the online and offline status of the bulb. You can change the device property value by setting the desired property value in IoT Platform.

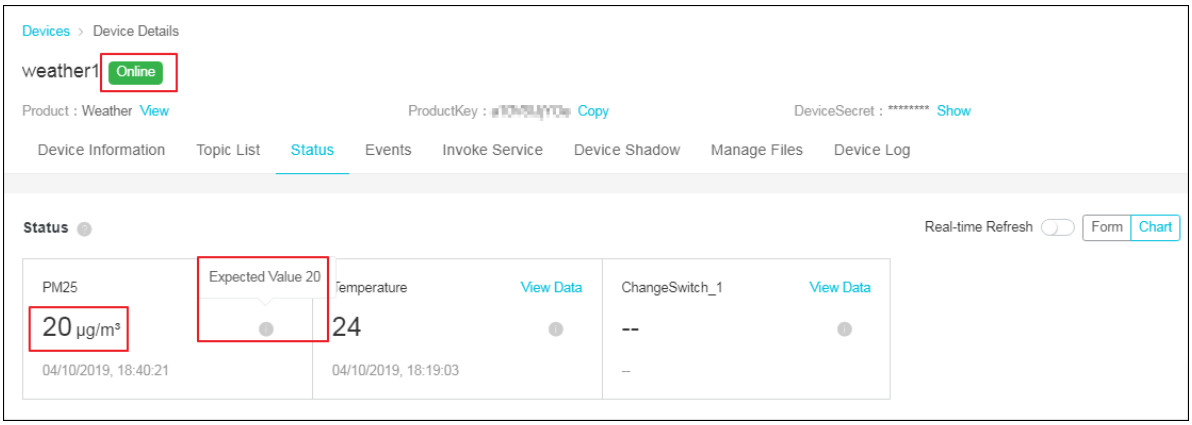
- If the bulb is online, you can change the bulb status in IoT Platform. The bulb responds to the change in real time.



- If the bulb is offline, you can also change the bulb status in IoT Platform. In this case, the desired property value in IoT Platform and the latest device property value are different.



- If the bulb goes from offline to online, it requests the desired property value. The latest property value is immediately synchronized to the desired value.



Appendix: Sample code to configure a device

```
package com.aliyun.alink.devicesdk.demo;
import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.JSONObject;
import com.aliyun.alink.apiclient.utils.StringUtils;
import com.aliyun.alink.dm.api.BaseInfo;
import com.aliyun.alink.dm.api.DeviceInfo;
import com.aliyun.alink.dm.api.InitResult;
import com.aliyun.alink.dm.model.RequestModel;
import com.aliyun.alink.dm.utils.IDGeneraterUtils;
```

```
import com.aliyun.alink.linkkit.api.ILinkKitConnectListener;
import com.aliyun.alink.linkkit.api.IoTMqttClientConfig;
import com.aliyun.alink.linkkit.api.LinkKit;
import com.aliyun.alink.linkkit.api.LinkKitInitParams;
import com.aliyun.alink.linksdk.cmp.api.ConnectSDK;
import com.aliyun.alink.linksdk.cmp.connect.channel.MqttPublishRequest;
import com.aliyun.alink.linksdk.cmp.core.base.ARequest;
import com.aliyun.alink.linksdk.cmp.core.base.AResponse;
import com.aliyun.alink.linksdk.cmp.core.listener.IConnectSendListener;
import com.aliyun.alink.linksdk.tmp.api.InputParams;
import com.aliyun.alink.linksdk.tmp.api.OutputParams;
import com.aliyun.alink.linksdk.tmp.device.payload.ValueWrapper;
import com.aliyun.alink.linksdk.tmp.devicemodel.Service;
import com.aliyun.alink.linksdk.tmp.listener.IPublishResourceListener;
import com.aliyun.alink.linksdk.tmp.listener.ITResRequestHandler;
import com.aliyun.alink.linksdk.tmp.listener.ITResResponseCallback;
import com.aliyun.alink.linksdk.tmp.utils.ErrorInfo;
import com.aliyun.alink.linksdk.tools.AError;
import com.aliyun.alink.linksdk.tools.ALog;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
public class LampDemo {
    private static final String TAG = "LampDemo";
    private final static String SERVICE_SET = "set";
    private final static String SERVICE_GET = "get";
    public static String DESIRED_PROPERTY_GET = "/sys/{productKey}/{deviceName}/thing/prope
rty/desired/get";
    public static String DESIRED_PROPERTY_GET_REPLY = "/sys/{productKey}/{deviceName}/thing
/property/desired/get_reply";
    public static String METHOD_GET_DESIRED_PROPERTY = "thing.property.desired.get";
    public static void main(String[] args) {
        /**
         * The certificate information about the device.
         */
        String productKey = "****";
        String deviceName = "Lamp";
        String deviceSecret = "*****";
        /**
         * The MQTT connection information.
         */
        String regionId = "cn-shanghai";
        LampDemo manager = new LampDemo();
        DeviceInfo deviceInfo = new DeviceInfo();
        deviceInfo.productKey = productKey;
        deviceInfo.deviceName = deviceName;
        deviceInfo.deviceSecret = deviceSecret;
        manager.init(deviceInfo, regionId);
    }
    public void init(final DeviceInfo deviceInfo, String region) {
        LinkKitInitParams params = new LinkKitInitParams();
        /**
         * Specify the parameters for MQTT initialization.
```

```
         */
        IoTMqttClientConfig config = new IoTMqttClientConfig();
        config.productKey = deviceInfo.productKey;
        config.deviceName = deviceInfo.deviceName;
        config.deviceSecret = deviceInfo.deviceSecret;
        config.channelHost = deviceInfo.productKey + ".iot-as-mqtt." + region + ".aliyuncs.
com:1883";
        /**
         * Specify whether to receive offline messages.
         * The cleanSession field that corresponds to the MQTT connection.
         */
        config.receiveOfflineMsg = false;
        params.mqttClientConfig = config;
        /**
         * Configure the initialization and pass in the certificate information about the d
evice.
         */
        params.deviceInfo = deviceInfo;
        LinkKit.getInstance().init(params, new ILinkKitConnectListener() {
            public void onError(AError aError) {
                ALog.e(TAG, "Init Error error=" + aError);
            }
            public void onInitDone(InitResult initResult) {
                ALog.i(TAG, "onInitDone result=" + initResult);
                connectNotifyListener();
                // Request the latest desired property value from IoT Platform.
                getDesiredProperty(deviceInfo, Arrays.asList("LightStatus"), new IConnectSe
ndListener() {
                    public void onResponse(ARequest aRequest, AResponse aResponse) {
                        if(aRequest instanceof MqttPublishRequest && aResponse.data != null
) {
                            JSONObject jsonObject = JSONObject.parseObject(aResponse.data.t
oString());
                            ALog.i(TAG, "onResponse result=" + jsonObject);
                            JSONObject dataObj = jsonObject.getJSONObject("data");
                            if (dataObj != null) {
                                if (dataObj.getJSONObject("LightStatus") == null) {
                                    // No desired value is set.
                                } else {
                                    Integer value = dataObj.getJSONObject("LightStatus").ge
tInteger("value");
                                    handlePropertySet("LightStatus", new ValueWrapper.IntVa
lueWrapper(value), true);
                                }
                            }
                        }
                    }
                    public void onFailure(ARequest aRequest, AError aError) {
                        ALog.d(TAG, "onFailure() called with: aRequest = [" + aRequest + "]
, aError = [" + aError + "]");
                    }
                });
            }
        });
```

```
    }
    /**
     * When the device handles a property value change, the methods are called in the follo
wing scenarios:
     * Scenario 1. Pull mode: After the device is connected to IoT Platform again, the devi
ce automatically requests the latest desired property value from IoT Platform.
     * Scenario 2. Push mode: When the device is online, it receives the desired property v
alues that IoT Platform pushes to the property.set topic.
     * @param identifier: the property identifier.
     * @param value: the desired property value.
     * @param needReport: specifies whether to report the property value to IoT Platform by
using the property.post topic.
     * In Scenario 2, the property report capability is integrated into the processing func
tion and the needReport parameter is set to false.
     * @return
     */
    private boolean handlePropertySet(String identifier, ValueWrapper value, boolean needRe
port) {
        ALog.d(TAG, "The device handles property changes= [" + identifier + "], value = ["
+ value + "]");
        // Check whether the property settings are successful based on the response. In thi
s example, a success message is returned.
        boolean success = true;
        if (needReport) {
            reportProperty(identifier, value);
        }
        return success;
    }
    private void reportProperty(String identifier, ValueWrapper value){
        if (StringUtils.isEmptyString(identifier) || value == null) {
            return;
        }
        ALog.d(TAG, "Report property identity=" + identifier);
        Map<String, ValueWrapper> reportData = new HashMap<>();
        reportData.put(identifier, value);
        LinkKit.getInstance().getDeviceThing().thingPropertyPost(reportData, new IPublishRe
sourceListener() {
            public void onSuccess(String s, Object o) {
                // The property value is reported.
                ALog.d(TAG, "Report success onSuccess() called with: s = [" + s + "], o = [
" + o + "]");
            }
            public void onError(String s, AError aError) {
                // The property value fails to be reported.
                ALog.d(TAG, "Report failure onError() called with: s = [" + s + "], aError
= [" + JSON.toJSONString(aError) + "]");
            }
        });
    }
    /**
     * Register a function to respond to service calls and property settings.
     * When IoT Platform calls a service from the device, the device must respond to the ca
ll and return a response.
     */
```

```
    public void connectNotifyListener() {
        List<Service> serviceList = LinkKit.getInstance().getDeviceThing().getServices();
        for (int i = 0; serviceList != null && i < serviceList.size(); i++) {
            Service service = serviceList.get(i);
            LinkKit.getInstance().getDeviceThing().setServiceHandler(service.getIdentifier(
), mCommonHandler);
        }
    }
    private ITResRequestHandler mCommonHandler = new ITResRequestHandler() {
        public void onProcess(String serviceIdentifier, Object result, ITResResponseCallbac
k itResResponseCallback) {
            ALog.d(TAG, "onProcess() called with: s = [" + serviceIdentifier + "]," +
                    " o = [" + result + "], itResResponseCallback = [" + itResResponseCallb
ack + "]");
            ALog.d(TAG, "Received an asynchronous service call from IoT Platform " + servic
eIdentifier);
            try {
                if (SERVICE_SET.equals(serviceIdentifier)) {
                    Map<String, ValueWrapper> data = (Map<String, ValueWrapper>)((InputPara
ms)result).getData();
                    ALog.d(TAG, "Received asynchronous downstream data " + data);
                    // Set the property value of the device and report the property value t
o IoT Platform.
                    boolean isSetPropertySuccess =
                            handlePropertySet("LightStatus", data.get("LightStatus"), false
);
                    if (isSetPropertySuccess) {
                        if (result instanceof InputParams) {
                            // Return a response to IoT Platform.
                            itResResponseCallback.onComplete(serviceIdentifier, null, null)
;
                        } else {
                            itResResponseCallback.onComplete(serviceIdentifier, null, null)
;
                        }
                    } else {
                        AError error = new AError();
                        error.setCode(100);
                        error.setMsg("setPropertyFailed.");
                        itResResponseCallback.onComplete(serviceIdentifier, new ErrorInfo(e
rror), null);
                    }
                } else if (SERVICE_GET.equals(serviceIdentifier)) {
                } else {
                    // The device operation varies by service.
                    ALog.d(TAG, "The returned values corresponding to the service.");
                    OutputParams outputParams = new OutputParams();
                    // outputParams.put("op", new ValueWrapper.IntValueWrapper(20));
                    itResResponseCallback.onComplete(serviceIdentifier, null, outputParams)
;
                }
            } catch (Exception e) {
                e.printStackTrace();
                ALog.d(TAG, "The format of the returned data is invalid");
            }
```

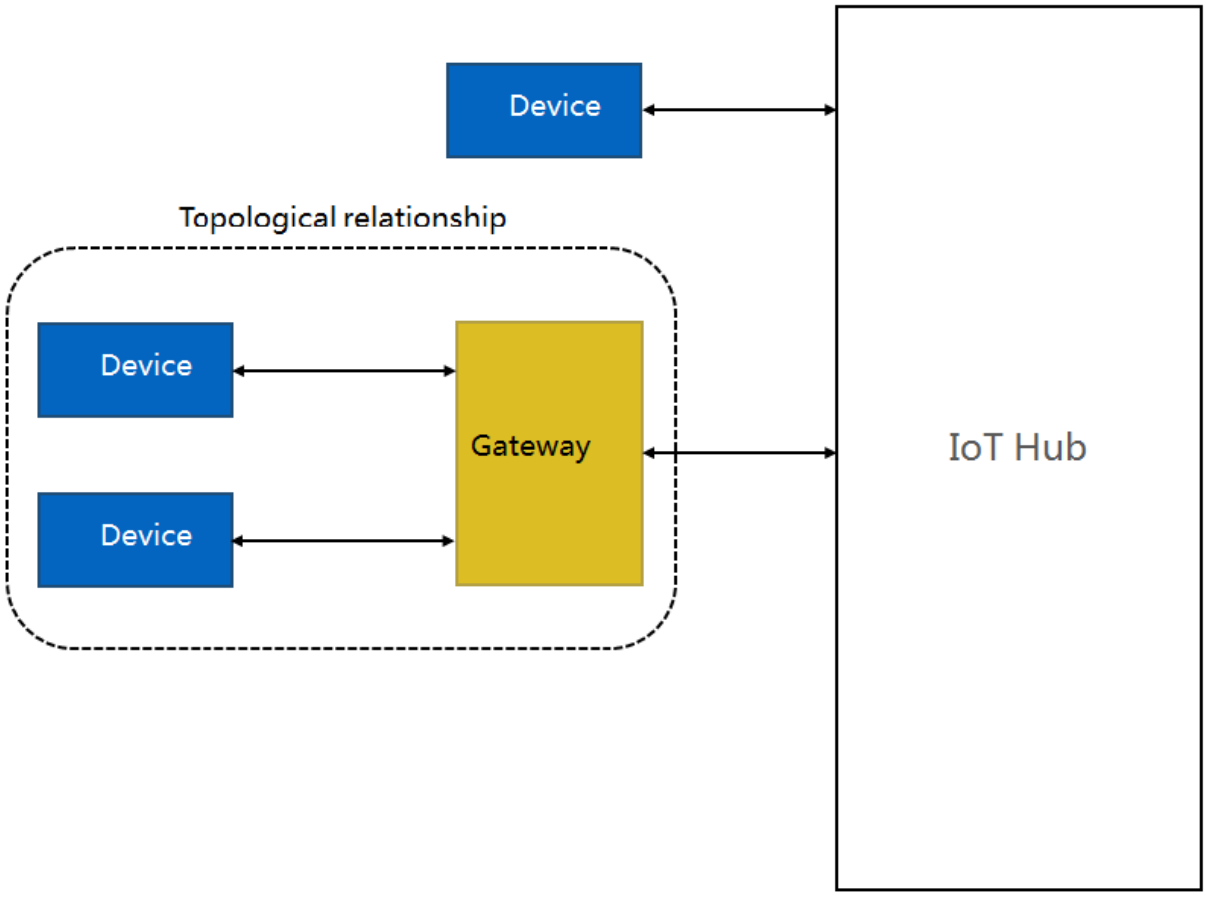
```
        }
        public void onSuccess(Object o, OutputParams outputParams) {
            ALog.d(TAG, "onSuccess() called with: o = [" + o + "], outputParams = [" + outp
utParams + "]");
            ALog.d(TAG, "The service was registered");
        }
        public void onFail(Object o, ErrorInfo errorInfo) {
            ALog.d(TAG, "onFail() called with: o = [" + o + "], errorInfo = [" + errorInfo
+ "]");
            ALog.d(TAG, "Service registration failed.");
        }
    };
    private void getDesiredProperty(BaseInfo info, List<String> properties, IConnectSendLis
tener listener) {
        ALog.d(TAG, "getDesiredProperty() called with: info = [" + info + "], listener = ["
+ listener + "]");
        if(info != null && !StringUtils.isEmptyString(info.productKey) && !StringUtils.isEm
ptyString(info.deviceName)) {
            MqttPublishRequest request = new MqttPublishRequest();
            request.topic = DESIRED_PROPERTY_GET.replace("{productKey}", info.productKey).r
eplace("{deviceName}", info.deviceName);
            request.replyTopic = DESIRED_PROPERTY_GET_REPLY.replace("{productKey}", info.pr
oductKey).replace("{deviceName}", info.deviceName);
            request.isRPC = true;
            RequestModel<List<String>> model = new RequestModel<>();
            model.id = String.valueOf(IDGeneraterUtils.getId());
            model.method = METHOD_GET_DESIRED_PROPERTY;
            model.params = properties;
            model.version = "1.0";
            request.payloadObj = model.toString();
            ALog.d(TAG, "getDesiredProperty: payloadObj=" + request.payloadObj);
            ConnectSDK.getInstance().send(request, listener);
        } else {
            ALog.w(TAG, "getDesiredProperty failed, baseInfo Empty.");
            if(listener != null) {
                AError error = new AError();
                error.setMsg("BaseInfoEmpty.");
                listener.onFailure(null, error);
            }
        }
    }
}
```

3.2. Connect a sub-device to IoT Platform

3.2.1. Overview

A sub-device can be connected to IoT Platform only by using a gateway. This article describes how to connect a sub-device to IoT Platform by using a gateway.

First, create a gateway and a sub-device in the IoT Platform console. Then, develop a device SDK for the gateway to connect the gateway to IoT Platform. Next, the gateway creates a topological relationship between the gateway and sub-devices and reports the relationship to IoT Platform. If a sub-device has a topological relationship with the gateway, the gateway reports the certificate of the sub-device to IoT Platform or dynamically registers the sub-device. IoT Platform validates the identity of the sub-device and the topological relationship between the sub-device and the gateway. If the validation is successful, IoT Platform establishes a logical channel between the sub-device and IoT Platform and associates the logical channel with the physical channel of the gateway. This way, the sub-device can connect to and communicate with IoT Platform by using the gateway.



This example describes only how to connect a sub-device to IoT Platform by using a gateway, and does not involve configurations for data communication such as Thing Specification Language (TSL) data communication. If you want to transmit TSL data between your sub-device and IoT Platform, configure your gateway to transmit the data for the sub-device. For more information, see Establish TSL-based communication.

> **Note** Data transmission between a gateway and a sub-device must be implemented by the provider of the gateway. Alibaba Cloud does not provide code for this feature.

Reference

You can download the sample code for connecting a sub-device to IoT Platform by using a gateway: iotx-api-demo.

To connect the sub-device to IoT Platform by using the gateway, perform the following steps:

1. Create a gateway and a sub-device

2. Initialize the SDKs
3. Connect the gateway device to IoT Platform
4. Connect the sub-device to IoT Platform

3.2.2. Create a gateway and a sub-device

This article describes how to create a gateway product, a gateway device, a sub-device product, and a sub-device.

Procedure

1. Log on to the IoT Platform console.
2.
3. In left-side navigation pane, choose **Devices > Products**.
4. On the Products page, click **Create Product** and create a gateway product and a sub-device product.
   - Create a gateway product. For more information, see Create a product.

     > **Note** Set the **Node Type** parameter to **Gateway device**.

   - Create a sub-device product. For more information, see Create a product.

     > **Note** Set the **Node Type** parameter to **Gateway sub-device**.

5. In the left-side navigation pane, choose Devices > **Devices**. On the **Devices** page, create a device for the gateway product and a sub-device for the sub-device product. For more information, see Create a device.

What's next

Initialize the SDKs

Connect the gateway device to IoT Platform

Connect the sub-device to IoT Platform

3.2.3. Initialize the SDKs

This article describes how to initialize IoT Platform SDK for Java and the device SDK for Java. You must prepare a Java development environment, download the SDK package, import a project, and then initialize the SDKs.

Prerequisites

Create a gateway and a sub-device

Procedure

1. Download and decompress the SDK package iotx-api-demo.

   This package contains the device SDK for Java and IoT Platform SDK for Java.

2. Import the decompressed SDK package to your Java development environment.

3. In the *pom* file in the *java* directory, add the following dependencies to import IoT Platform SDK for Java and the device SDK for Java.

```
<! -- https://mvnrepository.com/artifact/com.aliyun/aliyun-java-sdk-iot -->
<dependency>
    <groupId>com.aliyun</groupId>
    <artifactId>aliyun-java-sdk-iot</artifactId>
    <version>6.5.0</version>
</dependency>
<dependency>
    <groupId>com.aliyun</groupId>
    <artifactId>aliyun-java-sdk-core</artifactId>
    <version>3.5.1</version>
</dependency>
<dependency>
  <groupId>com.aliyun.alink.linksdk</groupId>
  <artifactId>iot-linkkit-java</artifactId>
  <version>1.2.0.1</version>
  <scope>compile</scope>
</dependency>
```

4. In the *config* file in the *java/src/main/resources/* directory, enter the required information for initialization.

```
user.accessKeyID = <your accessKey ID>
user.accessKeySecret = <your accessKey Secret>
iot.regionId = <regionId>
iot.productCode = Iot
iot.domain = iot.<regionId>.aliyuncs.com
iot.version = 2018-01-20
```

Parameter	Description
accessKeyID	The AccessKey ID of your Alibaba Cloud account. To create or view your AccessKey pair, perform the following steps: Log on to the Alibaba Cloud Management console. Move the pointer over your profile picture, and click **AccessKey Management** to go to the **AccessKey Pair** page.
accessKeySecret	The AccessKey secret of your Alibaba Cloud account. You can obtain the AccessKey secret the same way you obtain the AccessKey ID.
regionId	The ID of the region where your IoT devices reside. For more information about region IDs, see Regions and zones.

What's next

Connect the gateway device to IoT Platform

Connect the sub-device to IoT Platform

3.2.4. Connect the gateway device to IoT Platform

After you configure the device SDK, connect the gateway device to IoT Platform.

Prerequisites

Before you start, make sure that the following operations are performed:

- Create a gateway and a sub-device
- Initialize the SDKs

Configure the device SDK for the gateway device

In the SDK package, the *DeviceTopoManager* file in the *java/src/main/java/com/aliyun/iot/api/common/deviceApi* directory contains sample code for connecting the gateway device to IoT Platform.

1. Configure the information about the gateway device for connection.

```
private static String regionId = "cn-shanghai";
private static final String TAG = "TOPO";
// The information about the gateway device.
private static String GWproductKey = "a1BxptK***";
private static String GWdeviceName = "XMtrv3yvftEHAzrTfX1U";
private static String GWdeviceSecret = "19xJNybifnmgcK057vYhazYK4b64****";
public static void main(String[] args) {
    /**
     * The information about the MQTT connection.
     */
    DeviceTopoManager manager = new DeviceTopoManager();
    /**
     * The Java HTTP client of this device supports TSLv1.2.
     */
    System.setProperty("https.protocols", "TLSv2");
    manager.init();
}
```

2. Establish a connection.

```
public void init() {
        LinkKitInitParams params = new LinkKitInitParams();
        /**
         * Specify the parameters for MQTT initialization.
         */
        IoTMqttClientConfig config = new IoTMqttClientConfig();
        config.productKey = GWproductKey;
        config.deviceName = GWdeviceName;
        config.deviceSecret = GWdeviceSecret;
        config.channelHost = GWproductKey + ".iot-as-mqtt." + regionId + ".aliyuncs.com
:1883";
        /**
         * Specify whether to receive offline messages.
         * The cleanSession field that corresponds to the MQTT connection.
         */
        config.receiveOfflineMsg = false;
        params.mqttClientConfig = config;
        ALog.setLevel(LEVEL_DEBUG);
        ALog.i(TAG, "mqtt connetcion info=" + params);
        /**
         * Configure the initialization and pass in the certificate information about t
he device.
         */
        DeviceInfo deviceInfo = new DeviceInfo();
        deviceInfo.productKey = GWproductKey;
        deviceInfo.deviceName = GWdeviceName;
        deviceInfo.deviceSecret = GWdeviceSecret;
        params.deviceInfo = deviceInfo;
        /**Establish a connection. **/
        LinkKit.getInstance().init(params, new ILinkKitConnectListener() {
            public void onError(AError aError) {
                ALog.e(TAG, "Init Error error=" + aError);
            }
            public void onInitDone(InitResult initResult) {
                ALog.i(TAG, "onInitDone result=" + initResult);
                // Obtain the topological relationships of the gateway and check whethe
r a topological relationship exists between the gateway and the sub-device.
                // If a topological relationship exists between the gateway and the sub
-device, the gateway connects the sub-device to IoT Platform.
                getGWDeviceTopo();
                // Dynamically register the sub-device to obtain a DeviceSecret for the
sub-device. If the gateway has obtained the certificate of the sub-device, skip this st
ep.
                // When you register the sub-device, set the deviceName parameter to th
e serial number or MAC address of the sub-device.
                gatewaySubDevicRegister();
                //Specify the information about the sub-device for which you want to ad
d a topological relationship.
                gatewayAddSubDevice();
            }
        });
    }
```

Test the connection

After you configure the information about the gateway device in the device SDK, you can test the connection between the gateway device and IoT Platform.

1. Run DeviceTopoManager.
2. Log on to the IoT Platform console.In the left-side navigation pane, choose **Devices > Devices**.
3. On the Devices page, find the gateway device and check its status. If the status of the gateway device is **Online**, the gateway device is connected to IoT Platform.

What's next

Connect the sub-device to IoT Platform

3.2.5. Connect the sub-device to IoT Platform

A sub-device can be connected to IoT Platform only by using a gateway. After a sub-device is connected to a gateway, the gateway checks whether a topological relationship exists between the gateway and the sub-device. If so, the gateway reports the information about the sub-device to IoT Platform and connects the sub-device to IoT Platform.

Prerequisites

Before you start, make sure that the following operations are performed:

- Create a gateway and a sub-device
- Initialize the SDKs
- Connect the gateway device to IoT Platform

Context

- Configure a sub-device

  A sub-device cannot directly connect to IoT Platform. Therefore, you do not need to install a device SDK that IoT Platform provides on your sub-device. The device SDK of a sub-device must be developed by its provider.

- The SDK package provided by IoT Platform

  The *DeviceTopoManager* file in the *java/src/main/java/com/aliyun/iot/api/common/deviceApi* directory contains the code that is used by a gateway to manage topological relationships, obtain sub-device certificates, and connect a sub-device to IoT Platform by using the gateway.

Step 1: Manage topological relationships by using the gateway

After the gateway is connected to IoT Platform, the gateway must synchronize its topological relationships to IoT Platform. This way, the gateway can implement communication between sub-devices and IoT Platform. You can view and add topological relationships in the IoT Platform console. You can also perform these operations by using the sample code.

- View and add topological relationships between the gateway and sub-devices in the IoT Platform console.
  i. In the left-side navigation pane, choose **Devices > Devices**. On the Devices page, find the gateway device.
  ii. Click **Sub-device** in the Actions column. On the page that appears, view the sub-devices of the

gateway.

iii. Click **Add Sub-device**. Add the sub-device that you created in the Create a gateway and a sub-device step.

- Query and add topological relationships by using the following sample code.
  - Query topological relationships by using the following sample code:

```
     /**
     * Obtain the topological relationships of the gateway and check whether a topologi
cal relationship exists between the gateway and the sub-device.
     */
    private void getGWDeviceTopo() {
        LinkKit.getInstance().getGateway().gatewayGetSubDevices(new IConnectSendListene
r() {
            @Override
            public void onResponse(ARequest request, AResponse aResponse) {
                ALog.i(TAG, "The topological relationships of the gateway are obtained:
" + JSONObject.toJSONString(aResponse));
                // Obtain a list of sub-devices.
                try {
                    ResponseModel<List<DeviceInfo>> response = JSONObject.parseObject(a
Response.data.toString(), new TypeReference<ResponseModel<List<DeviceInfo>>>() {
                    }.getType());
                    // Process the request based on the actual scenario.
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
            @Override
            public void onFailure(ARequest request, AError error) {
                ALog.i(TAG, "Failed to obtain the topological relationships of the gate
way: " + JSONObject.toJSONString(error));
            }
        });
    }
```

  - Add a topological relationship by using the following sample code:

> **Note**
> - For information about how to obtain the certification information about the sub-device, see Step 2.
> - After IoT Platform validates the topological relationship between the gateway and the sub-device, the sub-device can use the physical channel of the gateway to communicate with IoT Platform.

```
    /**
     * Specify the information about the sub-device for which you want to add a topolog
ical relationship.
     */
    private void gatewayAddSubDevice() {
        BaseInfo baseInfo1 = new BaseInfo();
        baseInfo1.productKey = "a1j7SyR****";
```

```
        baseInfo1.deviceName = "safa***";
        String deviceSecret = "7lzCJIWHmGFpZpDKbJdVucDHUz6C****";
        LinkKit.getInstance().getGateway().gatewayAddSubDevice(baseInfo1, new ISubDevic
eConnectListener() {
            @Override
            public String getSignMethod() {
                // The signature method that you use to sign the request.
                return "hmacsha1";
            }
            @Override
            public String getSignValue() {
                // Generate a signature based on the DeviceSecret.
                Map<String, String> signMap = new HashMap<>();
                signMap.put("productKey", baseInfo1.productKey);
                signMap.put("deviceName", baseInfo1.deviceName);
                //signMap.put("timestamp", String.valueOf(System.currentTimeMillis()));
                signMap.put("clientId", getClientId());
                return SignUtils.hmacSign(signMap, deviceSecret);
            }
            @Override
            public String getClientId() {
                // Set the clientId parameter as needed.
                return "id";
            }
            @Override
            public Map<String, Object> getSignExtraData() {
                return null;
            }
            @Override
            public void onConnectResult(boolean isSuccess, ISubDeviceChannel iSubDevice
Channel, AError aError) {
                // Handle the result of the operation for adding the topological relati
onship between the sub-device and the gateway.
                if (isSuccess) {
                    // After the topological relationship is added, you can use the gat
eway to connect the sub-device to IoT Platform.
                    ALog.i(TAG, "The topological relationship is added: " + JSONObject.
toJSONString(iSubDeviceChannel));
                    // Connect the sub-device to IoT Platform by using the gateway.
                    gatewaySubDeviceLogin();
                } else {
                    ALog.i(TAG, "Failed to add the topological relationship: " + JSONOb
ject.toJSONString(aError));
                }
            }
            @Override
            public void onDataPush(String s, AMessage aMessage) {
            }
        });
    }
```

Step 2: Obtain the certificate information about the sub-device

After the sub-device is created, IoT Platform issues a certificate to the sub-device. The gateway can obtain the certificate information by using one of the following methods:

- Unique-certificate-per-device

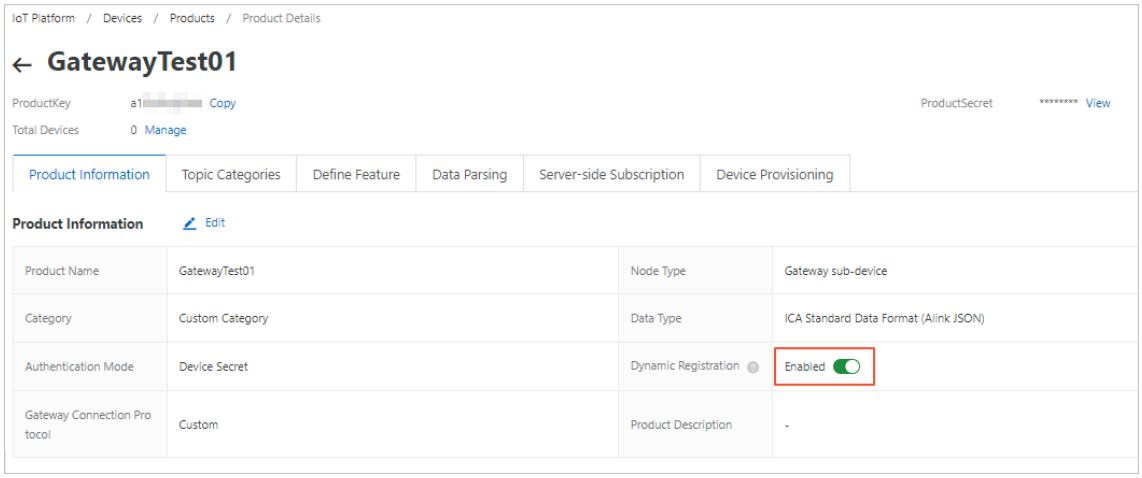
  You can obtain the ProductKey, DeviceName, and DeviceSecret of the sub-device on the details page of the sub-device in the IoT Platform console.

  - After a gateway discovers a connected sub-device, the gateway can obtain the certificate of the sub-device based on a protocol that is defined between the gateway and the sub-device. The protocol is defined by the provider of the gateway and the provider of the sub-device.
  - The provider of the gateway provides a configuration method that allows the gateway to preset certificate information about the sub-device. This feature is implemented by the provider of the gateway.

- Dynamic sub-device registration

  The gateway reports the ProductKey and DeviceName of the sub-device to IoT Platform to register the sub-device. After IoT Platform validates the ProductKey and DeviceName of the sub-device, IoT Platform dynamically assigns a DeviceSecret to the sub-device.

  i. When you create a sub-device in IoT Platform, set the deviceName parameter to the serial number or MAC address of the sub-device. After the sub-device is created, enable dynamic registration for the sub-device.



  ii. When you configure the gateway, make sure that the gateway can obtain the model and unique identifier, which is the serial number or MAC address, of the sub-device. This is made possible by the protocol that is defined between the gateway and the sub-device. In addition, map the sub-device model to the ProductKey in IoT Platform.

  iii. Obtain a DeviceSecret for the sub-device from IoT Platform during dynamic registration.

  Sample code:

```
    /**
     * Obtain a DeviceSecret for the sub-device during dynamic registration.
     * When you create the sub-device in IoT Platform, set the deviceName parameter to th
e serial number or MAC address of the sub-device.
     */
    private void gatewaySubDevicRegister() {
        List<BaseInfo> subDevices = new ArrayList<>();
        BaseInfo baseInfo1 = new BaseInfo();
        baseInfo1.productKey = "a1j7SyR***";
        baseInfo1.deviceName = "safasdf";
        subDevices.add(baseInfo1);
        LinkKit.getInstance().getGateway().gatewaySubDevicRegister(subDevices, new IConne
ctSendListener() {
            @Override
            public void onResponse(ARequest request, AResponse response) {
                ALog.i(TAG, "The sub-device is registered: " + JSONObject.toJSONString(re
sponse));
            }
            @Override
            public void onFailure(ARequest request, AError error) {
                ALog.i(TAG, "Failed to register the sub-device: " + JSONObject.toJSONStri
ng(error));
            }
        });
    }
```

For more information about dynamic registration of devices, see Overview.

Step 3: Connect the sub-device to IoT Platform by using the gateway.

```
    /**
     * Before you call an API operation to connect the sub-device to IoT Platform, make sur
e that a topological relationship is added between the sub-device and the gateway. After th
e gateway discovers a connected sub-device, the gateway reports the information about the s
ub-device to IoT Platform.
     * After the sub-device is connected to IoT Platform by using the gateway, the gateway
can subscribe to topics and publish messages for the sub-device.
     */
    public void gatewaySubDeviceLogin(){
        BaseInfo baseInfo1 = new BaseInfo();
        baseInfo1.productKey = "a1j7SyR****";
        baseInfo1.deviceName = "safasdf";
        LinkKit.getInstance().getGateway().gatewaySubDeviceLogin(baseInfo1, new ISubDeviceA
ctionListener() {
            @Override
            public void onSuccess() {
                // The gateway connects the sub-device to IoT Platform.
                // The gateway can subscribe to topics or publish messages for the sub-devi
ce. The gateway can also delete or disable the sub-device.
                // subDevDisable(null);
                // subDevDelete(null);
            }
            @Override
            public void onFailed(AError aError) {
                ALog.d(TAG, "onFailed() called with: aError = [" + aError + "]");
            }
        });
    }
}
```

Appendix: Sample code

A gateway discovers a connected sub-device and reports the information about the sub-device to IoT Platform. IoT Platform establishes a logical channel between the sub-device and IoT Platform. The sub-device uses the physical channel of the gateway to communicate with IoT Platform. The preceding process can be implemented in the following sample code:

```
package com.aliyun.iot.api.common.deviceApi;
import com.alibaba.fastjson.JSONObject;
import com.alibaba.fastjson.TypeReference;
import com.aliyun.alink.dm.api.BaseInfo;
import com.aliyun.alink.dm.api.DeviceInfo;
import com.aliyun.alink.dm.api.InitResult;
import com.aliyun.alink.dm.api.SignUtils;
import com.aliyun.alink.dm.model.ResponseModel;
import com.aliyun.alink.linkkit.api.ILinkKitConnectListener;
import com.aliyun.alink.linkkit.api.IoTMqttClientConfig;
import com.aliyun.alink.linkkit.api.LinkKit;
import com.aliyun.alink.linkkit.api.LinkKitInitParams;
import com.aliyun.alink.linksdk.channel.gateway.api.subdevice.ISubDeviceActionListener;
import com.aliyun.alink.linksdk.channel.gateway.api.subdevice.ISubDeviceChannel;
import com.aliyun.alink.linksdk.channel.gateway.api.subdevice.ISubDeviceConnectListener;
import com.aliyun.alink.linksdk.channel.gateway.api.subdevice.ISubDeviceRemoveListener;
```

```
import com.aliyun.alink.linksdk.cmp.core.base.AMessage;
import com.aliyun.alink.linksdk.cmp.core.base.ARequest;
import com.aliyun.alink.linksdk.cmp.core.base.AResponse;
import com.aliyun.alink.linksdk.cmp.core.listener.IConnectSendListener;
import com.aliyun.alink.linksdk.tools.AError;
import com.aliyun.alink.linksdk.tools.ALog;
import java.util.*;
import static com.aliyun.alink.linksdk.tools.ALog.LEVEL_DEBUG;
public class DeviceTopoManager {
    private static String regionId = "cn-shanghai";
    private static final String TAG = "TOPO";
    // The information about the gateway device.
    private static String GWproductKey = "a1Bxp*********";
    private static String GWdeviceName = "XMtrv3y**************";
    private static String GWdeviceSecret = "19xJNybifnmgc**************";
    public static void main(String[] args) {
        /**
         * The information about the MQTT connection.
         */
        DeviceTopoManager manager = new DeviceTopoManager();
        /**
         * The Java HTTP client of this device supports TSLv1.2.
         */
        System.setProperty("https.protocols", "TLSv2");
        manager.init();
    }
    public void init() {
        LinkKitInitParams params = new LinkKitInitParams();
        /**
         * Specify the parameters for MQTT initialization.
         */
        IoTMqttClientConfig config = new IoTMqttClientConfig();
        config.productKey = GWproductKey;
        config.deviceName = GWdeviceName;
        config.deviceSecret = GWdeviceSecret;
        config.channelHost = GWproductKey + ".iot-as-mqtt." + regionId + ".aliyuncs.com:188
3";
        /**
         * Specify whether to receive offline messages.
         * The cleanSession field that corresponds to the MQTT connection.
         */
        config.receiveOfflineMsg = false;
        params.mqttClientConfig = config;
        ALog.setLevel(LEVEL_DEBUG);
        ALog.i(TAG, "mqtt connetcion info=" + params);
        /**
         * Configure the initialization and pass in the certificate information about the g
ateway device.
         */
        DeviceInfo deviceInfo = new DeviceInfo();
        deviceInfo.productKey = GWproductKey;
        deviceInfo.deviceName = GWdeviceName;
        deviceInfo.deviceSecret = GWdeviceSecret;
        params.deviceInfo = deviceInfo;
        /**Establish a connection **/
```

```
        /* Establish a connection. */
        LinkKit.getInstance().init(params, new ILinkKitConnectListener() {
            public void onError(AError aError) {
                ALog.e(TAG, "Init Error error=" + aError);
            }
            public void onInitDone(InitResult initResult) {
                ALog.i(TAG, "onInitDone result=" + initResult);
                // Obtain the topological relationships of the gateway and check whether a
topological relationship exists between the gateway and the sub-device.
                // If a topological relationship exists between the gateway and the sub-dev
ice, the gateway connects the sub-device to IoT Platform.
                getGWDeviceTopo();
                // Dynamically register the sub-device to obtain a DeviceSecret for the sub
-device. If the gateway has obtained the certificate of the sub-device, skip this step.
                // When you create the sub-device in IoT Platform, set the deviceName param
eter to the serial number or MAC address of the sub-device.
                gatewaySubDevicRegister();
                //Specify the information about the sub-device for which you want to add a
topological relationship.
                gatewayAddSubDevice();
            }
        });
    }
    /**
     * Obtain the topological relationships of the gateway and check whether a topological
relationship exists between the gateway and the sub-device.
     */
    private void getGWDeviceTopo() {
        LinkKit.getInstance().getGateway().gatewayGetSubDevices(new IConnectSendListener()
{
            @Override
            public void onResponse(ARequest request, AResponse aResponse) {
                ALog.i(TAG, "The topological relationships of the gateway are obtained: " +
JSONObject.toJSONString(aResponse));
                // Obtain a list of sub-devices.
                try {
                    ResponseModel<List<DeviceInfo>> response = JSONObject.parseObject(aResp
onse.data.toString(), new TypeReference<ResponseModel<List<DeviceInfo>>>() {
                    }.getType());
                    // TODO. Write code based on your actual application scenario.
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
            @Override
            public void onFailure(ARequest request, AError error) {
                ALog.i(TAG, "Failed to obtain the topological relationships of the gateway:
" + JSONObject.toJSONString(error));
            }
        });
    }
    /**
     * Dynamically register the sub-device to obtain a DeviceSecret for the sub-device. If
the gateway has obtained the certificate of the sub-device, skip this step.
     * When you create the sub-device in IoT Platform, set the deviceName parameter to the
```

```
serial number or MAC address of the sub-device.
     */
    private void gatewaySubDevicRegister() {
        List<BaseInfo> subDevices = new ArrayList<>();
        BaseInfo baseInfo1 = new BaseInfo();
        baseInfo1.productKey = "a1j7SyR**********";
        baseInfo1.deviceName = "test123*********";
        subDevices.add(baseInfo1);
        LinkKit.getInstance().getGateway().gatewaySubDevicRegister(subDevices, new IConnect
SendListener() {
            @Override
            public void onResponse(ARequest request, AResponse response) {
                ALog.i(TAG, "The sub-device is registered: " + JSONObject.toJSONString(resp
onse));
            }
            @Override
            public void onFailure(ARequest request, AError error) {
                ALog.i(TAG, "Failed to register the sub-device: " + JSONObject.toJSONString
(error));
            }
        });
    }
    /**
     * Specify the information about the sub-device for which you want to add a topological
relationship.
     */
    private void gatewayAddSubDevice() {
        BaseInfo baseInfo1 = new BaseInfo();
        baseInfo1.productKey = "a1j7Sy*************";
        baseInfo1.deviceName = "safasd********";
        String deviceSecret = "7lzCJIWHmGF**************";
        LinkKit.getInstance().getGateway().gatewayAddSubDevice(baseInfo1, new ISubDeviceCon
nectListener() {
            @Override
            public String getSignMethod() {
                // The signature method that you use to sign the request.
                return "hmacsha1";
            }
            @Override
            public String getSignValue() {
                // Generate a signature based on the DeviceSecret.
                Map<String, String> signMap = new HashMap<>();
                signMap.put("productKey", baseInfo1.productKey);
                signMap.put("deviceName", baseInfo1.deviceName);
//                signMap.put("timestamp", String.valueOf(System.currentTimeMillis()));
                signMap.put("clientId", getClientId());
                return SignUtils.hmacSign(signMap, deviceSecret);
            }
            @Override
            public String getClientId() {
                // Set the clientId parameter as needed.
                return "id";
            }
            @Override
```

```
            public Map<String, Object> getSignExtraData() {
                return null;
            }
            @Override
            public void onConnectResult(boolean isSuccess, ISubDeviceChannel iSubDeviceChan
nel, AError aError) {
                // Handle the result of the operation for adding the topological relationsh
ip between the sub-device and the gateway.
                if (isSuccess) {
                    // After the topological relationship is added, you can use the gateway
to connect the sub-device to IoT Platform.
                    ALog.i(TAG, "The topological relationship is added: " + JSONObject.toJS
ONString(iSubDeviceChannel));
                    // Connect the sub-device to IoT Platform by using the gateway.
                    gatewaySubDeviceLogin();
                } else {
                    ALog.i(TAG, "Failed to add the topological relationship: " + JSONObject
.toJSONString(aError));
                }
            }
            @Override
            public void onDataPush(String s, AMessage aMessage) {
            }
        });
    }
    public void  gatewayDeleteSubDevice(){
        BaseInfo baseInfo1 = new BaseInfo();
        baseInfo1.productKey = "a1j7S**************";
        baseInfo1.deviceName = "saf*********";
        LinkKit.getInstance().getGateway().gatewayDeleteSubDevice(baseInfo1, new ISubDevice
RemoveListener() {
            @Override
            public void onSuceess() {
                // The sub-device is deleted. You can disconnect the sub-device before you
delete it.
            }
            @Override
            public void onFailed(AError aError) {
                // The sub-device failed to be deleted.
            }
        });
    }
    /**
     * Before you call an API operation to connect the sub-device to IoT Platform, make sur
e that a topological relationship is added between the sub-device and the gateway. After th
e gateway discovers a connected sub-device, the gateway reports the information about the s
ub-device to IoT Platform.
     * After the sub-device is connected to IoT Platform, the gateway can subscribe to topi
cs and publish messages for the sub-device.
     */
    public void gatewaySubDeviceLogin(){
        BaseInfo baseInfo1 = new BaseInfo();
        baseInfo1.productKey = "a1j7SyR**********";
        baseInfo1.deviceName = "safa*********";
```

```
        LinkKit.getInstance().getGateway().gatewaySubDeviceLogin(baseInfo1, new ISubDeviceA
ctionListener() {
            @Override
            public void onSuccess() {
                // The gateway connects the sub-device to IoT Platform.
                // The gateway can subscribe to topics or publish messages for the sub-devi
ce. The gateway can also delete or disable the sub-device.
                // subDevDisable(null);
                // subDevDelete(null);
            }
            @Override
            public void onFailed(AError aError) {
                ALog.d(TAG, "onFailed() called with: aError = [" + aError + "]");
            }
        });
    }
}
```

4.Maintenance and Monitoring

4.1. Use CloudMonitor to monitor IoT resources

4.1.1. Overview

You can use CloudMonitor to monitor IoT resources based on events and thresholds. Event monitoring and alerting are performed based on the throttling rules of IoT Platform. Threshold monitoring and alerting are performed based on the metric values that you have specified.

Context

Scenario

A total of 5,000 temperature sensors are deployed in an area that is designed to store hazardous goods. These devices communicate with IoT Platform by using the MQTT protocol. Every time the temperature fluctuates by 0.1°C, the temperature sensor sends the real-time temperature to IoT Platform. When the temperature exceeds 25°C, IoT Platform forwards the temperature data to Function Compute for subsequent processing.

Temperature data is an important factor that must be constantly monitored to ensure the safety of the storage area. The control room needs to obtain the temperature fluctuation data at each detection point in real time and respond in a timely manner. Therefore, the temperature sensors must be online 24 hours a day. If a large number of devices are offline at the same time, or the devices are unstable (repeatedly switching between online and offline), repair the devices and networks. A quick rise in temperature may pose a significant number of security risks that require an immediate response.

The temperature sensors at each detection point must be online 24 hours a day. Make sure that the data reported by the sensors can reach Function Compute and monitor the usage of IoT Platform resources under the current account. This helps to prevent throttling and ensure that IoT Platform can receive temperature data in a timely manner.

Project design

Before using CloudMonitor to monitor IoT Platform resources, create alert rules for IoT Platform in the CloudMonitor console.

Set the following events to trigger alerts:

- The number of connection requests that a device sends per minute reaches the maximum limit.
- The number of connection requests that the current account sends to IoT Platform per second reaches the maximum limit.
- The number of requests that the current account publishes per second reaches the maximum limit.
- The number of messages that a device sends per second reaches the maximum limit.
- The number of requests that the current account sends to the rules engine per second reaches the maximum limit.

Set the following thresholds to trigger alerts:

- The number of online devices (The MQTT protocol is used to connect the devices to IoT Platform).
- The number of property reporting failures.

- The number of messages that the rule engine forwards to Function Compute (FC).

Procedure for configuring alert rules

Create a product and devices

Configure alert contacts

Create event-triggered alert rules

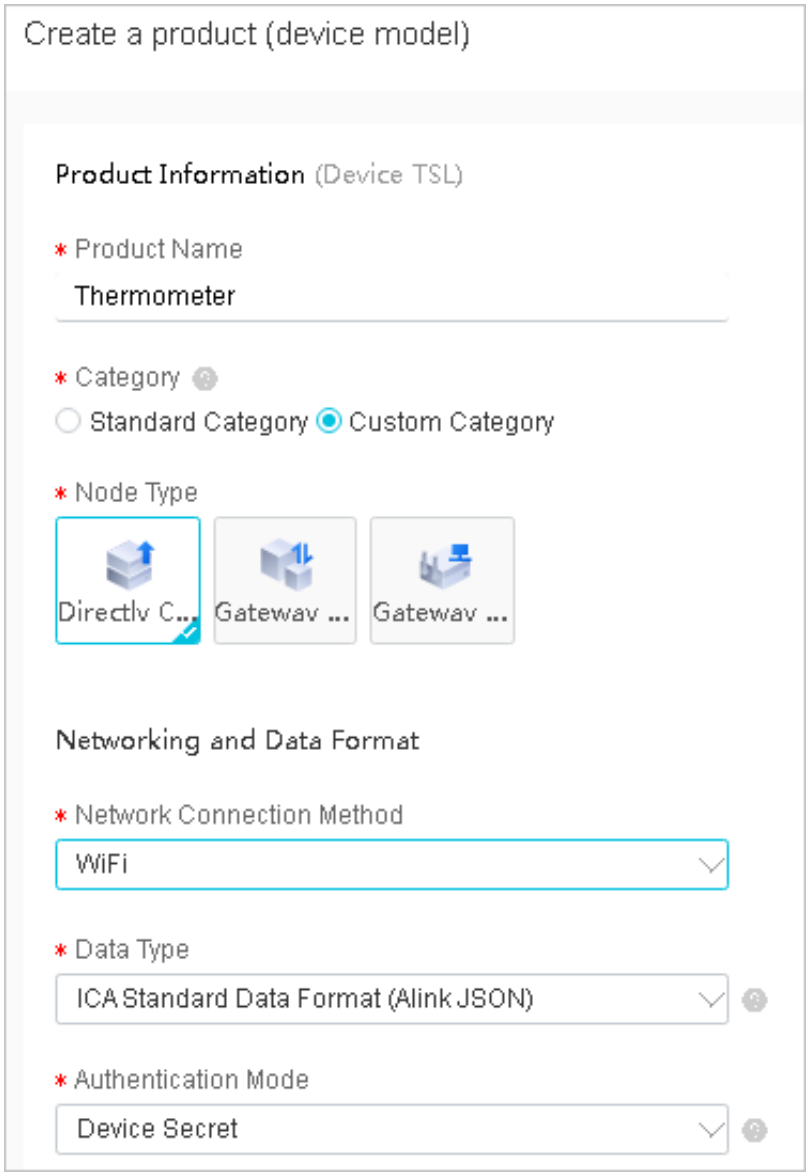
Create threshold-triggered alert rules

4.1.2. Create a product and devices

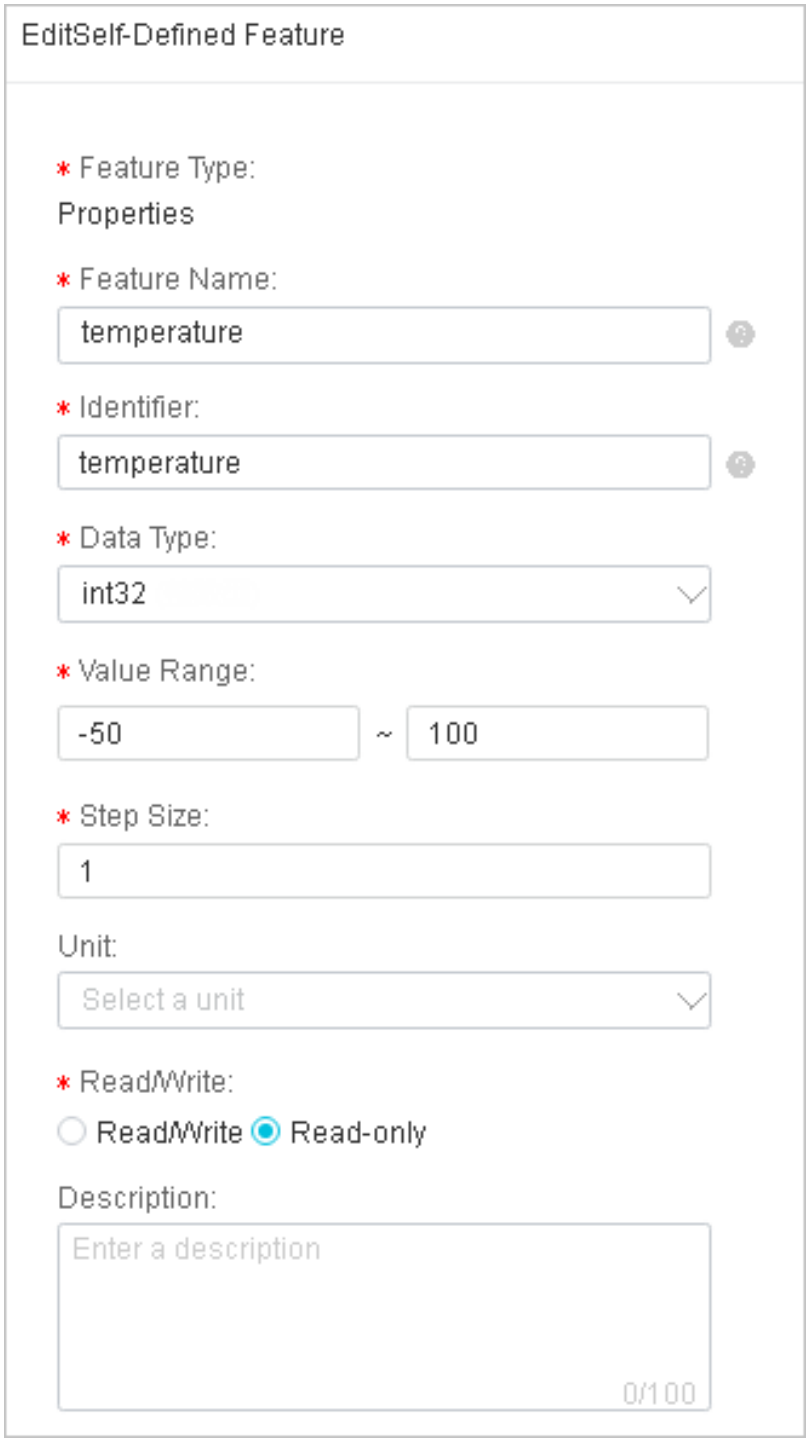
In this best practice, temperature sensors are monitored. This article describes how to create a temperature sensor product and devices in the IoT Platform console. This topic also describes how to define a Thing Specification Language (TSL) model and create a data forwarding rule.

Procedure

1. Log on to the IoT Platform console.
2.
3. In the left-side navigation pane, choose **Devices > Products**. On the page that appears, click **Create Product** to create a product. For more information, see Create a product.



4. Define a feature.
   i. On the **Product Details** page, click the **Define Feature** tab. Click **Edit Draft** and then **Add Self-defined Feature** to define a temperature property for the product.



ii. After the feature is added, click **Release online** to publish the TSL model.

5. In the left-side navigation pane, choose **Devices** > **Devices**. On the page that appears, click **Batch Add** to add multiple devices.

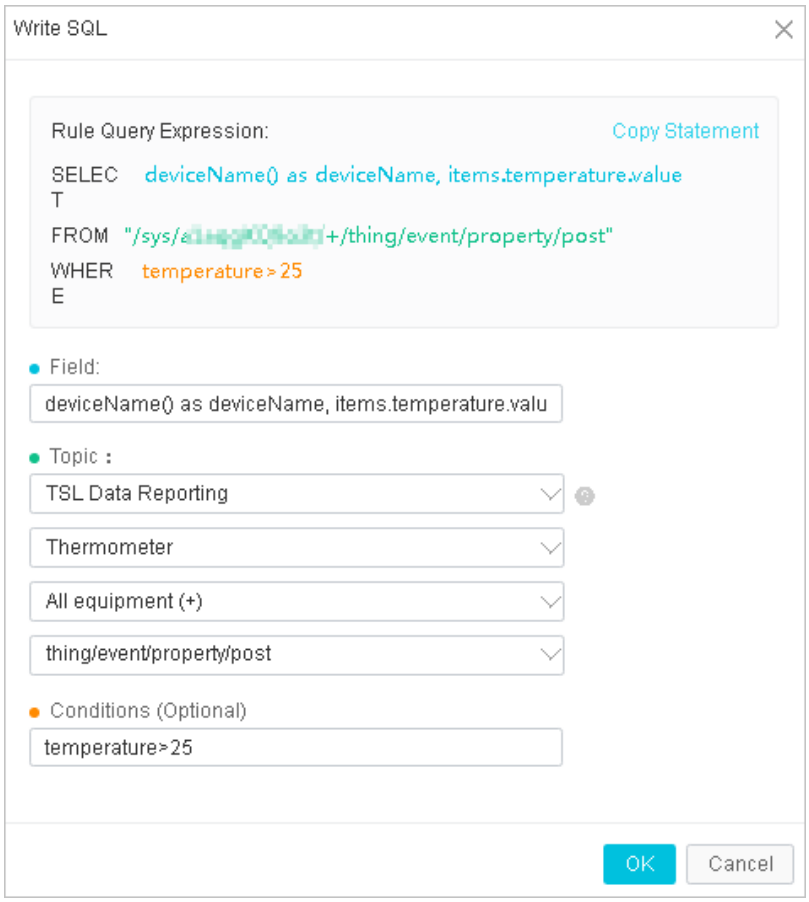
   In this example, you need to add 5,000 devices. You can add a maximum of 1,000 devices at a time. Therefore, you must add devices a minimum of five times. For more information, see Create multiple devices at a time.

6. In the left-side navigation pane, choose **Rules** > **Data Forwarding**. On the page that appears, create a data forwarding rule to allow forwarding of the temperature data reported by the temperature sensor to Function Compute.

i. Click **Create Rule** to create a rule that processes JSON-formatted data.

ii. Click **View** to go to the details page of the rule and write an SQL statement to process data.

The SQL statement in this example filters data from the reported property data. The SQL statement returns the values of the deviceName and Temperature fields when the temperature is higher than 25°C. The two fields are forwarded to Function Compute.



iii. Add a data forwarding operation and set the data forwarding destination to Function Compute.

iv. After the rule configuration is complete, return to the **Data Forwarding** page and click **Start** to enable the rule.

For more information, see Configure a data forwarding rule.

What's next

Configure alert contacts

Create event-triggered alert rules

Create threshold-triggered alert rules

4.1.3. Configure alert contacts

If Cloud Monitor detects an exception, it sends an alert notification to an alert group by using an email or DingTalk chatbot message. You must create an alert contact and add the contact to the alert group. When you create an alert rule, select the alert group to receive alert notifications.

Prerequisites

To receive alerts by using a DingTalk chatbot, you must create a DingTalk chatbot (an intelligent assistant for a DingTalk group) in the DingTalk application. For more information, see Receive alert notifications from a DingTalk group.

Procedure

1. Create an alert contact. For more information, see Create an alert contact.
2. Create an alert group and add the alert contact to the alert group. For more information, see Create an alert contact group.

   You can also add alert contacts to an existing alert group. For more information, see Add multiple alert contacts to an alert contact group at a time.

What's next

Create event-triggered alert rules

Create threshold-triggered alert rules

4.1.4. Create event-triggered alert rules

IoT Platform imposes limits on the frequency of connection requests, the frequency of upstream and downstream communication, and the frequency of message forwarding. When the limits are reached, throttling is triggered, and this will have an adverse effect on the daily operation of your business. With the monitoring and alerting functions of CloudMonitor, you can receive alert notifications to perform corresponding business adjustments in a timely manner.

Context

For IoT Platform usage restrictions, see Limits.

Procedure

1. Log on to the CloudMonitor console.
2. In the left-side navigation pane, choose **Alarms > Alarm Rules**.
3. On the **Alarm Rules** page, choose **Event Alarm > Create Event Alert**.
4. In the **Create/Modify Event Alert** dialog box that appears on the right side of the page, enter a rule name, configure an alert rule, and then select an alert group and a notification method. Click **OK**.

   The following table describes how to configure an event-triggered alert rule.

Parameter	Description
Alarm Rule Name	Follow the instructions to set a valid rule name.
Event Type	Select **System Event**.
Product Type	Select **IoT Platform**.
Event Type	Select **All types**.
Event Level	Select **All Levels**.

Parameter	Description
Event Name	Select the events that you want to monitor. ○ Account_Connect_QPS_Limit ○ Device_Connect_QPM_Limit ○ Account_Uplink_QPS_Limit ○ Device_Uplink_QPS_Limit ○ Account_RuleEngine_DataForward_QPS_Limit
Resource Range	Select **All Resources**.
Alarm Type	Select **Alert Notification**. Select an alert group and a notification method.

5. Test the alert rule.
   i. On the **Alarm Rules** page, find the alert rule and click **test** in the Actions column.
   ii. In the **Create event test** dialog box that appears on the right side of the page, select an event name, modify the alert notification content, and then click **OK** to run a test on the alert rule.

   After you select an event name, the JSON-formatted alert notification content is displayed in the text box below. You can modify the notification content.
   iii. Check that the contacts of the alert group can receive the modified alert notification.

   For example, members in a DingTalk group will receive a message that is similar to the following.

What's next

Create threshold-triggered alert rules

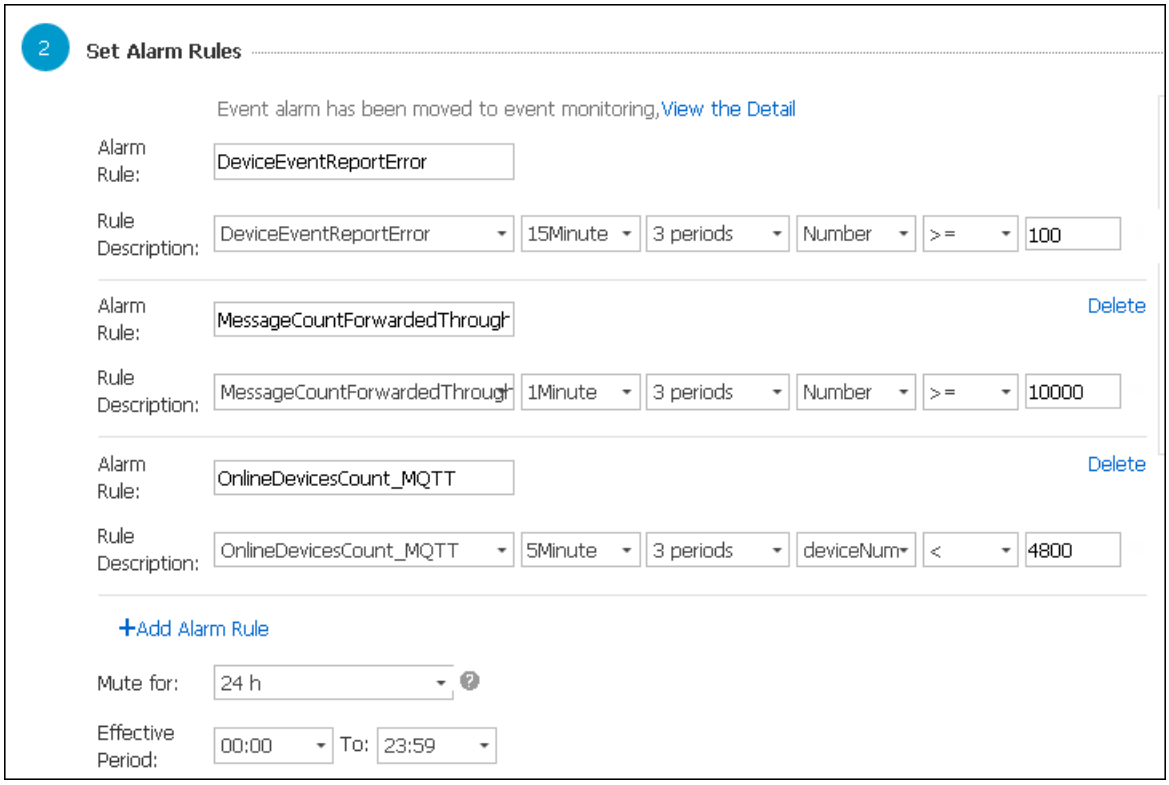
4.1.5. Create threshold-triggered alert rules

You can create threshold-triggered alert rules in CloudMonitor to monitor IoT resources, and send alert notifications to specific contacts by using the specified method. You can monitor the number of online devices that are under a specific product, the number of TSL model communication failures, and the number of times that data is forwarded by the rule engine.

Procedure

1. Log on to the CloudMonitor console.
2. In the left-side navigation pane, choose **Alarms > Alarm Rules**.
3. On the **Threshold Value Alarm** tab of the **Alarm Rules** page, click **Create Alarm Rule**.
4. Select IoT Platform as the related resource and specify the product to be monitored.
5. Set alert rules. Add the required alert rules.

   In this example, three threshold-triggered alert rules are set, and the alert mute period is set to 10 minutes. (If an alert is not cleared 10 minutes after the alert is triggered, the alert notification will be sent again.)



The following table describes the rules shown in the preceding figure.

Rule name	Rule description	Description
DevicePropertyReportError	The statistical period is 15 minutes. An alert is triggered if a minimum of 100 property reporting failures are detected for three consecutive statistical periods.	This rule is used to monitor the property reporting status of all devices under the temperature-based sensory product. If a large number of consecutive property reporting failures occur, you must check the devices and network status.
MessageCountForwardedThroughRuleEngine_FC	The statistical period is 1 minute. An alert is triggered if a minimum of 10,000 messages are forwarded to Function Compute for three consecutive statistical periods.	According to the design, when the temperature reported by a device is higher than 25°C, the rules engine forwards the temperature data to Function Compute. If 10,000 messages are forwarded, a large number of devices have reported more than two temperatures that are above 25°C within 1 minute. We recommend that you monitor the temperature change and perform early troubleshooting of potential safety risks.
OnlineDevicesCount_MQTT	The statistical period is 5 minutes. An alert is triggered if less than 4,800 devices are online for three consecutive statistical periods.	A large number of devices are offline at the same time. The devices and network status must be checked.

6. Set the alert notification method. Select an alert group and a notification method. You can specify or configure the other fields.
7. Click **Confirm** to complete rule creation.

Result

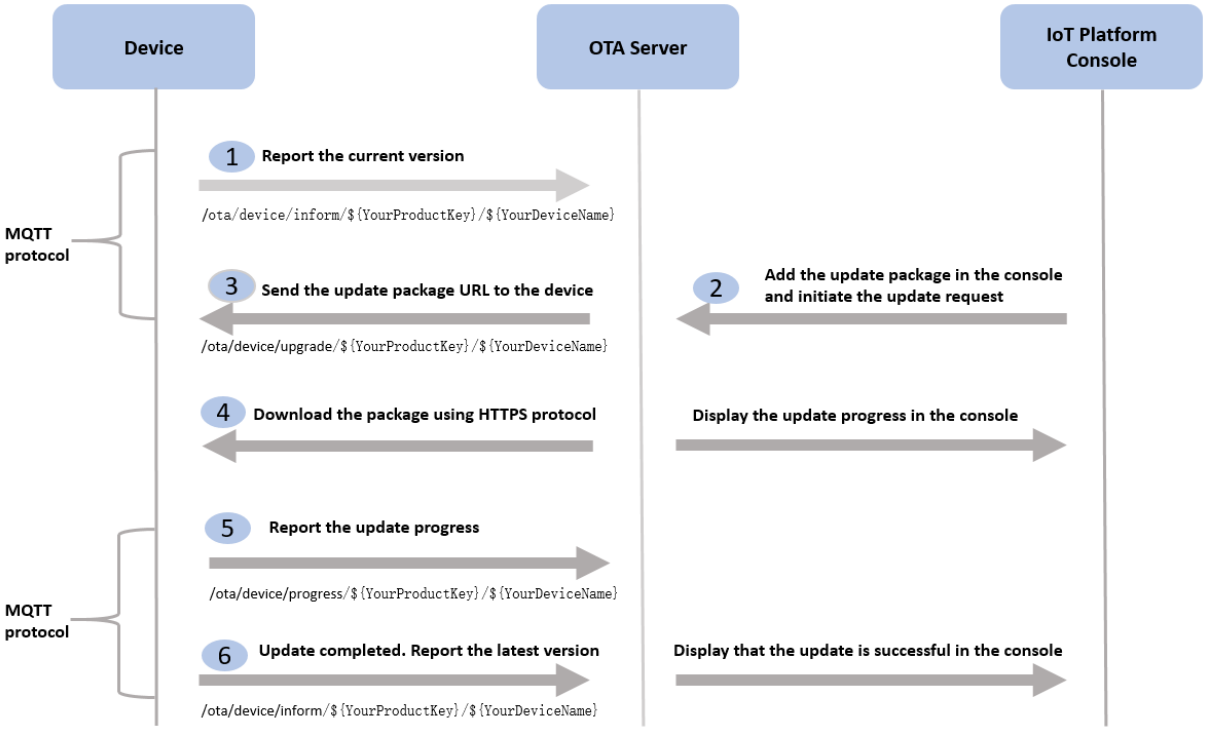
After a threshold-triggered alert rule is created, CloudMonitor continuously monitors IoT Platform resources according to the rule. When an alert rule is triggered, CloudMonitor sends alert notifications by using the specified method.

4.2. Configure OTA updates for devices

4.2.1. Overview

Over-the-Air Technology (OTA) is a basic feature of IoT Platform. OTA allows you to update the firmware of IoT devices worldwide. This chapter uses an example to describe the OTA update process. Sample codes are also provided for configuring OTA updates on devices.

OTA update process for device firmware



1. (Optional) The device reports the current firmware version to the topic: `/ota/device/inform/${YourProductKey}/${YourDeviceName}`.

```
{
  "id": 1,
  "params": {
    "version": "1-0-0"
  }
}
```

2. The device subscribes to the topic: `/ota/device/upgrade/${YourProductKey}/${YourDeviceName}` where IoT Platform pushes OTA notifications.

   The format of update notifications:

```
{
    "code":"1000",
    "data":{
        "size":11472299,
        "sign":"83254ac96e141affb8aa42cbfec9****",
        "version":"2-0-0",
        "url":"https://iotx-ota.oss-cn-shanghai.aliyuncs.com/ota/dbab6f742ae389b40db88f
c2500b****/ck0q5lyav00003i7hezxe****.zip?Expires=1568951190&OSSAccessKeyId=cS8uRRy54Rsz
****&Signature=nk0sogaxtyp7dYvKZnjNQ%2BZ8Q9****",
        "signMethod":"Md5",
        "md5":"83254ac96e141affb8aa42cbfec9****"
    },
    "id":1568864790381,
    "message":"success"
}
```

3. The device downloads the firmware package from the URL that is provided in the update notification and performs a local update.
4. The device reports the update progress to the topic: `/ota/device/progress/${YourProductKey}/${YourDeviceName}`.

   The format of report messages:

```
{
  "id": 1,
  "params": {
    "step":"1",
    "desc":" xxxxxxxx "
  }
}
```

5. The device reports the updated firmware version to the topic: `/ota/device/inform/${YourProductKey}/${YourDeviceName}`.

   The format of report messages:

```
{
  "id": 1,
  "params": {
    "version": "2-0-0"
  }
}
```

References

Configure OTA update for devices

Push firmware files to devices

4.2.2. Configure OTA update for devices

You must enable over-the-air (OTA) update when you develop devices. When you prepare an update file, you can update the original firmware configurations and compile the update file into a file format that is supported by IoT Platform. This article provides the sample code that shows how to configure OTA update for a device.

Prerequisites

A product and devices are created in the IoT Platform console. The device certificate is obtained.

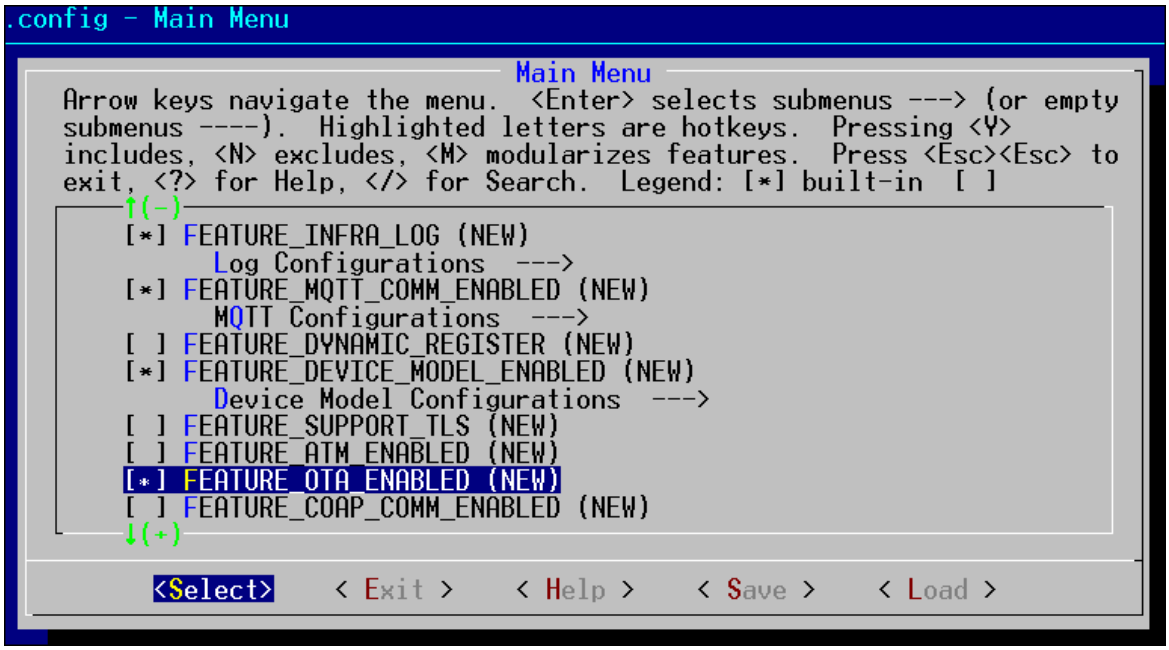
Context

In this example, Link SDK V3.x for C provided by Alibaba Cloud is used to develop devices. For more information about how to download the SDK demo, visit Link SDK.

In the SDK demo, *fota* indicates OTA update.

Connect a device to IoT Platform

1. Run the **make menuconfig** command to go to the compilation settings page. In compilation settings, select FEATURE_OTA_ENABLED (NEW) to enable OTA update for the device.



2. Go to the *src/dev_model/examples* directory and open the *linkkit_example_solo.c* sample file.

   The sample file contains the following code for OTA update.

```
/** fota event handler **/
static int user_fota_event_handler(int type, const char *version)
{
    char buffer[128] = {0};
    int buffer_length = 128;
    /* 0 - new firmware exist, query the new firmware */
    if (type == 0) {
        EXAMPLE_TRACE("New Firmware Version: %s", version);
        IOT_Linkkit_Query(EXAMPLE_MASTER_DEVID, ITM_MSG_QUERY_FOTA_DATA, (unsigned char
*)buffer, buffer_length);
    }
    return 0;
}
```

3. Edit the device information in the *linkkit_example_solo.c* file.

```
char g_product_key[IOTX_PRODUCT_KEY_LEN + 1]       = "a1G0LGy****";
char g_product_secret[IOTX_PRODUCT_SECRET_LEN + 1] = "UdW1LOv7QiMm****";
char g_device_name[IOTX_DEVICE_NAME_LEN + 1]       = "test01";
char g_device_secret[IOTX_DEVICE_SECRET_LEN + 1]   = "rtpyII8e9H97C7BGizYQs010htRx***";
```

4. Run the program to connect the device to IoT Platform.

Update firmware configurations

1. Edit the *linkkit_example_solo.c* file to update the configurations based on your workload requirements.

   For more information about the update file in this example, see the sample update file.

2. Compile the updated *linkkit_example_solo.c* file and save the compiled BIN file.

   In this example, the saved BIN file is named *app_1001*.

   In Push firmware files to devices, you need to add an update package in the IoT Platform console and upload the BIN file.

Sample update file

```
/*
 * Copyright (C) 2015-2018 Alibaba Group Holding Limited
 */
#include "infra_config.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#ifdef __UBUNTU_SDK_DEMO__
    #include <unistd.h>
#endif
#include "infra_types.h"
#include "infra_defs.h"
#include "infra_compat.h"
#ifdef INFRA_MEM_STATS
    #include "infra_mem_stats.h"
#endif
```

```
#include "dev_model_api.h"
#include "wrappers.h"
#include "cJSON.h"
#ifdef ATM_ENABLED
    #include "at_api.h"
#endif
//#include "bind_api.h"
char g_product_key[IOTX_PRODUCT_KEY_LEN + 1]       = "a1G0LGy****";
char g_product_secret[IOTX_PRODUCT_SECRET_LEN + 1] = "UdW1LOv7QiMm****";
char g_device_name[IOTX_DEVICE_NAME_LEN + 1]       = "teset01";
char g_device_secret[IOTX_DEVICE_SECRET_LEN + 1]   = "rtpyII8e9H97C7BGizYQs010htRx****";
#define EXAMPLE_TRACE(...)                                          \
    do {                                                            \
        HAL_Printf("\033[1;32;40m%s.%d: ", __func__, __LINE__);     \
        HAL_Printf(__VA_ARGS__);                                    \
        HAL_Printf("\033[0m\r\n");                                  \
    } while (0)
void run_ubuntu_wifi_provision_example()
{
#ifdef __UBUNTU_SDK_DEMO__
#if defined(WIFI_PROVISION_ENABLED)
    char *wifi_name = "linkkit";
    char buffer[128] = {0};
    int ret;
    extern int awss_config_press();
    extern int awss_start();
    memset(buffer, 0, 128);
    snprintf(buffer, 128, "nmcli connection down %s", wifi_name);
    ret = system(buffer);
    memset(buffer, 0, 128);
    snprintf(buffer, 128, "nmcli connection delete %s", wifi_name);
    ret = system(buffer);
    sleep(15);
    awss_config_press();
    awss_start();
#endif
#endif
}
#define EXAMPLE_MASTER_DEVID            (0)
#define EXAMPLE_YIELD_TIMEOUT_MS        (200)
typedef struct {
    int master_devid;
    int cloud_connected;
    int master_initialized;
} user_example_ctx_t;
static user_example_ctx_t g_user_example_ctx;
static user_example_ctx_t *user_example_get_ctx(void)
{
    return &g_user_example_ctx;
}
void user_post_raw_data(void)
{
    int res = 0;
    user_example_ctx_t *user_example_ctx = user_example_get_ctx();
```

```
    unsigned char raw_data[2] = {0x01, 0x02};
    res = IOT_Linkkit_Report(user_example_ctx->master_devid, ITM_MSG_POST_RAW_DATA,
                             raw_data, 7);
    EXAMPLE_TRACE("Post Raw Data Message ID: %d", res);
}
static int user_down_raw_data_arrived_event_handler(const int devid, const unsigned char *p
ayload,
        const int payload_len)
{
    EXAMPLE_TRACE("Down Raw Message, Devid: %d, Payload Length: %d", devid, payload_len);
   // user_post_raw_data();
    return 0;
}
/** cloud connected event callback */
static int user_connected_event_handler(void)
{
    EXAMPLE_TRACE("Cloud Connected");
    g_user_example_ctx.cloud_connected = 1;
    return 0;
}
/** cloud disconnected event callback */
static int user_disconnected_event_handler(void)
{
    EXAMPLE_TRACE("Cloud Disconnected");
    g_user_example_ctx.cloud_connected = 0;
    return 0;
}
/* device initialized event callback */
static int user_initialized(const int devid)
{
    EXAMPLE_TRACE("Device Initialized");
    g_user_example_ctx.master_initialized = 1;
    return 0;
}
/** recv property post response message from cloud **/
static int user_report_reply_event_handler(const int devid, const int msgid, const int code
, const char *reply,
        const int reply_len)
{
    EXAMPLE_TRACE("Message Post Reply Received, Message ID: %d, Code: %d, Reply: %.*s", msg
id, code,
                  reply_len,
                  (reply == NULL) ? ("NULL") : (reply));
    return 0;
}
/** recv event post response message from cloud **/
static int user_trigger_event_reply_event_handler(const int devid, const int msgid, const i
nt code, const char *eventid,
        const int eventid_len, const char *message, const int message_len)
{
    EXAMPLE_TRACE("Trigger Event Reply Received, Message ID: %d, Code: %d, EventID: %.*s, M
essage: %.*s",
                  msgid, code,
                  eventid_len,
```

```
                    eventid, message_len, message);
    return 0;
}
/** recv property setting message from cloud **/
static int user_property_set_event_handler(const int devid, const char *request, const int
request_len)
{
    int res = 0;
    EXAMPLE_TRACE("Property Set Received, Request: %s", request);
    res = IOT_Linkkit_Report(EXAMPLE_MASTER_DEVID, ITM_MSG_POST_PROPERTY,
                             (unsigned char *)request, request_len);
    EXAMPLE_TRACE("Post Property Message ID: %d", res);
    return 0;
}
static int user_service_request_event_handler(const int devid, const char *serviceid, const
int serviceid_len,
        const char *request, const int request_len,
        char **response, int *response_len)
{
    int add_result = 0;
    cJSON *root = NULL, *item_number_a = NULL, *item_number_b = NULL;
    const char *response_fmt = "{\"Result\": %d}";
    EXAMPLE_TRACE("Service Request Received, Service ID: %.*s, Payload: %s", serviceid_len,
serviceid, request);
    /* Parse Root */
    root = cJSON_Parse(request);
    if (root == NULL || ! cJSON_IsObject(root)) {
        EXAMPLE_TRACE("JSON Parse Error");
        return -1;
    }
    if (strlen("Operation_Service") == serviceid_len && memcmp("Operation_Service", service
id, serviceid_len) == 0) {
        /* Parse NumberA */
        item_number_a = cJSON_GetObjectItem(root, "NumberA");
        if (item_number_a == NULL || ! cJSON_IsNumber(item_number_a)) {
            cJSON_Delete(root);
            return -1;
        }
        EXAMPLE_TRACE("NumberA = %d", item_number_a->valueint);
        /* Parse NumberB */
        item_number_b = cJSON_GetObjectItem(root, "NumberB");
        if (item_number_b == NULL || ! cJSON_IsNumber(item_number_b)) {
            cJSON_Delete(root);
            return -1;
        }
        EXAMPLE_TRACE("NumberB = %d", item_number_b->valueint);
        add_result = item_number_a->valueint + item_number_b->valueint;
        /* Send Service Response To Cloud */
        *response_len = strlen(response_fmt) + 10 + 1;
        *response = (char *)HAL_Malloc(*response_len);
        if (*response == NULL) {
            EXAMPLE_TRACE("Memory Not Enough");
            return -1;
        }
        memset(*response, 0, *response_len);
```

```
        memset(*response, 0, *response_len);
        HAL_Snprintf(*response, *response_len, response_fmt, add_result);
        *response_len = strlen(*response);
    }
    cJSON_Delete(root);
    return 0;
}
static int user_timestamp_reply_event_handler(const char *timestamp)
{
    EXAMPLE_TRACE("Current Timestamp: %s", timestamp);
    return 0;
}
/** fota event handler **/
static int user_fota_event_handler(int type, const char *version)
{
    char buffer[128] = {0};
    int buffer_length = 128;
    /* 0 - new firmware exist, query the new firmware */
    if (type == 0) {
        EXAMPLE_TRACE("New Firmware Version: %s", version);
        IOT_Linkkit_Query(EXAMPLE_MASTER_DEVID, ITM_MSG_QUERY_FOTA_DATA, (unsigned char *)b
uffer, buffer_length);
    }
    return 0;
}
/* cota event handler */
static int user_cota_event_handler(int type, const char *config_id, int config_size, const
char *get_type,
                                   const char *sign, const char *sign_method, const char *u
rl)
{
    char buffer[128] = {0};
    int buffer_length = 128;
    /* type = 0, new config exist, query the new config */
    if (type == 0) {
        EXAMPLE_TRACE("New Config ID: %s", config_id);
        EXAMPLE_TRACE("New Config Size: %d", config_size);
        EXAMPLE_TRACE("New Config Type: %s", get_type);
        EXAMPLE_TRACE("New Config Sign: %s", sign);
        EXAMPLE_TRACE("New Config Sign Method: %s", sign_method);
        EXAMPLE_TRACE("New Config URL: %s", url);
        IOT_Linkkit_Query(EXAMPLE_MASTER_DEVID, ITM_MSG_QUERY_COTA_DATA, (unsigned char *)b
uffer, buffer_length);
    }
    return 0;
}
void user_post_property(void)
{
    static int cnt = 0;
    int res = 0;
    char property_payload[30] = {0};
    HAL_Snprintf(property_payload, sizeof(property_payload), "{\"Counter\": %d}", cnt++);
    res = IOT_Linkkit_Report(EXAMPLE_MASTER_DEVID, ITM_MSG_POST_PROPERTY,
                             (unsigned char *)property_payload, strlen(property_payload));
    EXAMPLE_TRACE("Post Property Message ID: %d", res);
```

```
}
void user_post_event(void)
{
    int res = 0;
    char *event_id = "HardwareError";
    char *event_payload = "{\"ErrorCode\": 0}";
    res = IOT_Linkkit_TriggerEvent(EXAMPLE_MASTER_DEVID, event_id, strlen(event_id),
                                   event_payload, strlen(event_payload));
    EXAMPLE_TRACE("Post Event Message ID: %d", res);
}
void user_deviceinfo_update(void)
{
    int res = 0;
    char *device_info_update = "[{\"attrKey\":\"abc\",\"attrValue\":\"hello,world\"}]";
    res = IOT_Linkkit_Report(EXAMPLE_MASTER_DEVID, ITM_MSG_DEVICEINFO_UPDATE,
                             (unsigned char *)device_info_update, strlen(device_info_update
));
    EXAMPLE_TRACE("Device Info Update Message ID: %d", res);
}
void user_deviceinfo_delete(void)
{
    int res = 0;
    char *device_info_delete = "[{\"attrKey\":\"abc\"}]";
    res = IOT_Linkkit_Report(EXAMPLE_MASTER_DEVID, ITM_MSG_DEVICEINFO_DELETE,
                             (unsigned char *)device_info_delete, strlen(device_info_delete
));
    EXAMPLE_TRACE("Device Info Delete Message ID: %d", res);
}
static int user_cloud_error_handler(const int code, const char *data, const char *detail)
{
    EXAMPLE_TRACE("code =%d ,data=%s, detail=%s", code, data, detail);
    return 0;
}
static int dynreg_device_secret(const char *device_secret)
{
    EXAMPLE_TRACE("device secret: %s", device_secret);
    return 0;
}
static int user_sdk_state_dump(int ev, const char *msg)
{
    printf("received state: -0x%04X(%s)\n", -ev, msg);
    return 0;
}
int main(int argc, char **argv)
{
    int res = 0;
    int cnt = 0;
    int auto_quit = 0;
    iotx_linkkit_dev_meta_info_t master_meta_info;
    int domain_type = 0, dynamic_register = 0, post_reply_need = 0, fota_timeout = 30;
#ifdef ATM_ENABLED
    if (IOT_ATM_Init() < 0) {
        EXAMPLE_TRACE("IOT ATM init failed! \n");
        return -1;
```

```
    }
#endif
    if (argc >= 2 && ! strcmp("auto_quit", argv[1])) {
        auto_quit = 1;
        cnt = 0;
    }
    memset(&g_user_example_ctx, 0, sizeof(user_example_ctx_t));
    memset(&master_meta_info, 0, sizeof(iotx_linkkit_dev_meta_info_t));
    memcpy(master_meta_info.product_key, g_product_key, strlen(g_product_key));
    memcpy(master_meta_info.product_secret, g_product_secret, strlen(g_product_secret));
    memcpy(master_meta_info.device_name, g_device_name, strlen(g_device_name));
    memcpy(master_meta_info.device_secret, g_device_secret, strlen(g_device_secret));
    IOT_SetLogLevel(IOT_LOG_DEBUG);
    /*
     * if the following conditions are met:
     *    1) wifi provision is enabled,
     *    2) current OS is Ubuntu,
     *    3) a wireless card is inserted,
     *    4) g_ifname in HAL_AWSS_linux.c has been set to be the wireless card's name accor
ding to ifconfig,
     *       for example, the output of command "ifconfig" is like:
     *         wlx00259ce0**** Link encap:Ethernet  HWaddr 00:25:9c:e0:**:**
     *         UP BROADCAST PROMISC MULTICAST  MTU:1500  Metric:1
     *         RX packets:8709 errors:0 dropped:27 overruns:0 frame:0
     *         TX packets:2457 errors:0 dropped:0 overruns:0 carrier:0
     *         collisions:0 txqueuelen:1000
     *         RX bytes:2940097 (2.9 MB)  TX bytes:382827 (382.8 KB)
     *       then set g_ifname = "00259ce0****"
     *    5) the linkkit-example-solo is running with sudo permission.
     *  Then you can run wifi-provision example in Ubuntu, just to uncomment the following
line.
     */
    /* Register Callback */
    IOT_RegisterCallback(ITE_STATE_EVERYTHING, user_sdk_state_dump);
    IOT_RegisterCallback(ITE_CONNECT_SUCC, user_connected_event_handler);
    IOT_RegisterCallback(ITE_DISCONNECTED, user_disconnected_event_handler);
    IOT_RegisterCallback(ITE_SERVICE_REQUEST, user_service_request_event_handler);
    IOT_RegisterCallback(ITE_PROPERTY_SET, user_property_set_event_handler);
    IOT_RegisterCallback(ITE_REPORT_REPLY, user_report_reply_event_handler);
    IOT_RegisterCallback(ITE_TRIGGER_EVENT_REPLY, user_trigger_event_reply_event_handler);
    IOT_RegisterCallback(ITE_TIMESTAMP_REPLY, user_timestamp_reply_event_handler);
    IOT_RegisterCallback(ITE_INITIALIZE_COMPLETED, user_initialized);
    IOT_RegisterCallback(ITE_FOTA, user_fota_event_handler);
    IOT_RegisterCallback(ITE_COTA, user_cota_event_handler);
    IOT_RegisterCallback(ITE_CLOUD_ERROR, user_cloud_error_handler);
    IOT_RegisterCallback(ITE_DYNREG_DEVICE_SECRET, dynreg_device_secret);
    IOT_RegisterCallback(ITE_RAWDATA_ARRIVED, user_down_raw_data_arrived_event_handler);
    domain_type = IOTX_CLOUD_REGION_SHANGHAI;
    IOT_Ioctl(IOTX_IOCTL_SET_DOMAIN, (void *)&domain_type);
    /* Choose Login Method */
    dynamic_register = 0;
    IOT_Ioctl(IOTX_IOCTL_SET_DYNAMIC_REGISTER, (void *)&dynamic_register);
    /* post reply doesn't need */
    post_reply_need = 1;
```

```
    IOT_Ioctl(IOTX_IOCTL_RECV_EVENT_REPLY, (void *)&post_reply_need);
    IOT_Ioctl(IOTX_IOCTL_FOTA_TIMEOUT_MS, (void *)&fota_timeout);
    do {
        g_user_example_ctx.master_devid = IOT_Linkkit_Open(IOTX_LINKKIT_DEV_TYPE_MASTER, &m
aster_meta_info);
        if (g_user_example_ctx.master_devid >= 0) {
            break;
        }
        EXAMPLE_TRACE("IOT_Linkkit_Open failed! retry after %d ms\n", 2000);
        HAL_SleepMs(2000);
    } while (1);
    /* run_ubuntu_wifi_provision_example(); */
    do {
        res = IOT_Linkkit_Connect(g_user_example_ctx.master_devid);
        if (res >= 0) {
            break;
        }
        EXAMPLE_TRACE("IOT_Linkkit_Connect failed! retry after %d ms\n", 5000);
        HAL_SleepMs(5000);
    } while (1);
    while (1) {
        IOT_Linkkit_Yield(EXAMPLE_YIELD_TIMEOUT_MS);
        /* Post Proprety Example */
        if ((cnt % 2) == 0) {
            /* user_post_property(); */
        }
        /* Post Event Example */
        if ((cnt % 10) == 0) {
            //    user_post_event();
        }
        cnt++;
        if((cnt % 20)==0)
        {
     //       IOT_Bind_Reset();
        }
        if (auto_quit == 1 && cnt > 3600) {
            break;
        }
    }
    IOT_Linkkit_Close(g_user_example_ctx.master_devid);
    IOT_DumpMemoryStats(IOT_LOG_DEBUG);
    IOT_SetLogLevel(IOT_LOG_NONE);
    return 0;
}
```

4.2.3. Push firmware files to devices

This article describes how to push firmware files to devices by using the IoT Platform console. To complete the process, you must perform the following operations: add an update package, verify the update package, and start a batch update.

Prerequisites

- The Perform OTA updates feature is supported on the devices to be updated.

  Only devices for which the OTA service is enabled can report firmware versions, receive update messages from IoT Platform, download firmware, and perform OTA update operations.

- A firmware file is generated based on your business requirements. For more information about generation methods and sample files, see Configure OTA update for devices.

Procedure

1. Log on to the IoT Platform console.
2. In the left-side navigation pane, choose **Maintenance > OTA Update**.
3. On the **OTA Update** page, click **Add Update Package**.
4. Set the required parameters, upload the firmware file as the update package, and then click **OK**.

   The following table describes some parameters. For more information about how to set other parameters, see Overview.

Parameter	Value
Types of Update Packages	Full
Update Package Module	default
Update Package Version	2-0-0
Signature Algorithm	MD5

5. On the Update Package tab, click **Verify** in the Actions column of the update package. IoT Platform verifies the update package by using a device. For more information, see Overview. If the device is updated, it indicates that the verification succeeds. Then, the **Batch Update** button becomes available.
6. Click **Batch Update**, configure the required update scope and update policy, and then click OK to push update notifications to devices in batches. For more information, see Overview.

View device logs

After IoT Platform pushes OTA update notifications, you can view the information about the OTA update from the logs of the devices. The logs include the notifications, processing details, firmware information, update progress, and reporting time.

- The following log example includes an update notification that is received by a device.

```
[1630984617.566][LK-0309] pub: /ota/device/upgrade/g[redacted][/SDevice1

[LK-030A] < 7B 22 63 6F 64 65 22 3A  22 31 30 30 30 22 2C 22 | {"code":"1000","
[LK-030A] < 64 61 74 61 22 3A 7B 22  73 69 7A 65 22 3A 31 35 | data":{"size":15
[LK-030A] < 30 32 35 2C 22 73 69 67  6E 22 3A 22 39 65 61 35 | 025,"sign":"9ea5
[LK-030A] < 36 34 33 36 61 63 37 38  61 39 31 32 63 32 38 64 | [redacted]
[LK-030A] < 30 64 66 61 36 31 64 62  66 35 31 63 22 2C 22 76 | 0dfa61dbf51c","v
[LK-030A] < 65 72 73 69 6F 6E 22 3A  22 32 2E 30 2E 30 22 2C | ersion":"2.0.0",
[LK-030A] < 22 73 69 67 6E 4D 65 74  68 6F 64 22 3A 22 4D 64 | "signMethod":"Md
[LK-030A] < 35 22 2C 22 75 72 6C 22  3A 22 68 74 74 70 73 3A | 5","url":"https:
[LK-030A] < 2F 2F 69 6F 74 78 2D 6F  74 61 2E 6F 73 73 2D 63 | //iotx-ota.oss-c
[LK-030A] < 6E 2D 73 68 61 6E 67 68  61 69 2E 61 6C 69 79 75 | n[redacted]u
[LK-030A] < 6E 63 73 2E 63 6F 6D 2F  6F 74 61 2F 62 63 64 36 | n[redacted]6
[LK-030A] < 31 34 32 35 39 34 64 30  31 38 33 61 31 36 64 38 | 1[redacted]8
[LK-030A] < 32 35 61 64 38 32 32 35  35 34 64 34 2F 63 6B 74 | 2[redacted]t
[LK-030A] < 39 68 34 79 67 6D 30 30  30 30 33 62 38 66 72 6B | 9[redacted]k
[LK-030A] < 69 6E 61 6F 79 39 2E 62  69 6E 3F 45 78 70 69 72 | i[redacted]r
[LK-030A] < 65 73 3D 31 36 33 31 30  37 31 30 31 37 26 4F 53 | e[redacted]S
[LK-030A] < 53 41 63 63 65 73 73 4B  65 79 49 64 3D 4C 54 41 | S[redacted]A
[LK-030A] < 49 34 47 31 54 75 57 77  53 69 72 6E 62 41 7A 55 | I[redacted]U
[LK-030A] < 48 66 4C 33 65 26 53 69  67 6E 61 74 75 72 65 3D | H[redacted]=
[LK-030A] < 49 25 32 46 4E 4F 32 4F  6E 54 58 66 58 6D 6D 35 | I[redacted]5
[LK-030A] < 45 76 47 52 77 30 35 64  47 4E 51 6F 55 25 33 44 | E[redacted]D
[LK-030A] < 22 2C 22 6D 64 35 22 3A  22 39 65 61 35 36 34 33 | ","md5":"9ea5643
[LK-030A] < 36 61 63 37 38 61 39 31  32 63 32 38 64 30 64 66 | [redacted]
[LK-030A] < 61 36 31 64 62 66 35 31  63 22 7D 2C 22 69 64 22 | a61dbf51c"},"id"
[LK-030A] < 3A 31 36 33 30 39 38 34  36 31 37 35 35 33 2C 22 | :1630984617553,"
[LK-030A] < 6D 65 73 73 61 67 65 22  3A 22 73 75 63 63 65 73 | message":"succes
[LK-030A] < 73 22 7D                                        | s"}
```

- The following log example includes the information of a new firmware version. The log example also includes the details about how a device connects to the download link of the firmware.

```
OTA target firmware version: 2.0.0, size: 15025 Bytes
starting download thread in 2 seconds ......
core_sysdep_network_establish host iotx-[redacted].aliyuncs.com port 443, type 0
establish tcp connection with server(host='iotx-[redacted].aliyuncs.com', port=[443])
success to establish tcp, fd=4
local port: 58960
[1630984619.566][LK-1000] establish mbedtls connection with server(host='io[redacted]liyuncs.com', port=[443])
[1630984619.622][LK-1000] success to establish mbedtls connection,  (cost 47168 bytes in total, max used 49856 bytes)
[1630984619.622][LK-040B] > GET /ota/b[redacted]9.bin?Expires=16[redacted]7&OSSAccessKeyId=LT[redacted]S
[1630984619.622][LK-040B] > Host: iotx-[redacted] aliyuncs.com
[1630984619.622][LK-040B] > Accept: text/html, application/xhtml+xml, application/xml;q=0.9, */*;q=0.8
[1630984619.622][LK-040B] > Range: bytes=0-
[1630984619.622][LK-040B] > Content-Length: 0
[1630984619.622][LK-040B] >
```

- The following log example includes the details about how a device downloads a firmware file and reports the progress.

```
[1630984619.622][LK-0309] pub: /ota/device/progress/[redacted]I/SDevice1

[LK-030A] > 7B 22 69 64 22 3A 32 2C  20 22 70 61 72 61 6D 73 | {"id":2, "params
[LK-030A] > 22 3A 7B 22 73 74 65 70  22 3A 22 30 22 2C 22 64 | ":{"step":"0","d
[LK-030A] > 65 73 63 22 3A 22 22 7D  7D                      | esc":""}}

[1630984619.644][LK-040D] < HTTP/1.1 206 Partial Content
[1630984619.644][LK-040D] < Server: AliyunOSS
[1630984619.644][LK-040D] < Date: Tue, 07 Sep 2021 03:16:59 GMT
[1630984619.644][LK-040D] < Content-Type: application/octet-stream
[1630984619.644][LK-040D] < Content-Length: 15025
[1630984619.644][LK-040D] < Connection: keep-alive
[1630984619.644][LK-040D] < x-oss-request-id: 6136D9AB3731FB36345F2350
[1630984619.644][LK-040D] < Content-Range: bytes 0-15024/15025
[1630984619.644][LK-040D] < Accept-Ranges: bytes
[1630984619.644][LK-040D] < ETag: "9EA56[redacted]"
[1630984619.644][LK-040D] < Last-Modified: Tue, 07 Sep 2021 02:47:02 GMT
[1630984619.644][LK-040D] < x-oss-object-type: Normal
[1630984619.644][LK-040D] < x-oss-hash-crc64ecma: 1317[redacted]
[1630984619.644][LK-040D] < x-oss-storage-class: Standard
[1630984619.644][LK-040D] < Content-MD5: r[redacted]
[1630984619.644][LK-040D] < x-oss-server-time: 16
[1630984619.644][LK-040D] <
download 054% done, +8192 bytes
[1630984619.644][LK-0309] pub: /ota/device/progress/g1[redacted]I/SDevice1

[LK-030A] > 7B 22 69 64 22 3A 33 2C  20 22 70 61 72 61 6D 73 | {"id":3, "params
[LK-030A] > 22 3A 7B 22 73 74 65 70  22 3A 22 35 34 22 2C 22 | ":{"step":"54","
[LK-030A] > 64 65 73 63 22 3A 22 22  7D 7D                   | desc":""}}

[1630984619.644][LK-0901] digest matched
download 100% done, +6833 bytes
[1630984619.644][LK-0309] pub: /ota/device/progress/g[redacted]I/SDevice1

[LK-030A] > 7B 22 69 64 22 3A 34 2C  20 22 70 61 72 61 6D 73 | {"id":4, "params
[LK-030A] > 22 3A 7B 22 73 74 65 70  22 3A 22 31 30 30 22 2C | ":{"step":"100",
[LK-030A] > 22 64 65 73 63 22 3A 22  22 7D 7D                | "desc":""}}

download completed
```

5.Data forwarding

5.1. Use the Alibaba Cloud big data platform to create a dashboard to monitor devices

This topic describes how to integrate IoT Platform with the Alibaba Cloud big data platform. This way, you can use the Alibaba Cloud big data platform to compute and analyze device data, collect the statistics of devices, and display device data, statistics, and analysis results in real time.

Prerequisites

- The related Alibaba Cloud services are activated and the required instances and computing resources are purchased. When you use the Alibaba Cloud big data platform to process the device data of IoT Platform, you may need to use multiple Alibaba Cloud services, such as ApsaraDB RDS for MySQL and IoT Platform.
- You are familiar with the configuration procedure and usage notes for the Alibaba Cloud services.

Context

- Scenario: Use a DataV dashboard to display the data of the devices of an IoT Platform product in real time.
- Process:
  i. IoT Platform collects device data.
  ii. IoT Platform forwards device data of a product to DataHub by using the rules engine.
  iii. DataHub sends the device data to Realtime Compute for Apache Flink for processing and writes the processed data to ApsaraDB RDS for MySQL. If the device data does not need to be processed, you can synchronize the data from DataHub to ApsaraDB RDS for MySQL by using DataConnector.
  iv. DataV uses an ApsaraDB RDS for MySQL database table as a data source and displays device data in the database table in real time.

Procedure

1. Create an ApsaraDB RDS for MySQL database to store device data.

   For more information, see ApsaraDB RDS for MySQL.

   i. Log on to the ApsaraDB RDS console.
   ii. On the **Instances** page, click **Create Instance** to purchase an ApsaraDB RDS for MySQL instance.

   > **Note** The region where the ApsaraDB RDS for MySQL instance resides must be the same as the region where your IoT Platform devices and DataHub project reside.

   iii. On the Instances page, find the purchased instance and click **Manage** in the Actions column.

iv. In the left-side navigation pane, click **Accounts**. On the Accounts page, click Create Account and configure the parameters.

v. In the left-side navigation pane, click **Databases**. On the Databases page, click Create Database and configure the parameters.

vi. In the left-side navigation pane, click **Data Security**. On the Data Security page, click Create Whitelist to configure an IP address whitelist for the ApsaraDB RDS for MySQL instance. For more information, see Configure an IP address whitelist for an ApsaraDB RDS for MySQL instance.

vii. In the left-side navigation pane, click **Basic Information** to view the basic information about the instance.

If you want to synchronize data to DataHub, DataV, or Realtime Compute for Apache Flink, the instance information is required.

viii. In the top navigation bar of the **Basic Information** page, click **Log On to Database**. In the Login instance dialog box, enter the information about an account to log on to the database.

ix. Tables are created in the databases. In this example, a database table named mytable is created. The database table contains two fields. The following table describes the fields.

mytable

Field	Type	Description
d_data	varchar(32)	The time.
device_num	int	The number of active devices.

2. In the IoT Platform console, create a product, a device, and a data forwarding rule.

i. Log on to the IoT Platform console.

ii.

iii. In the left-side navigation pane, choose **Devices > Products**. On the Products page, click Create Product and configure the parameters. For more information, see Create a product.

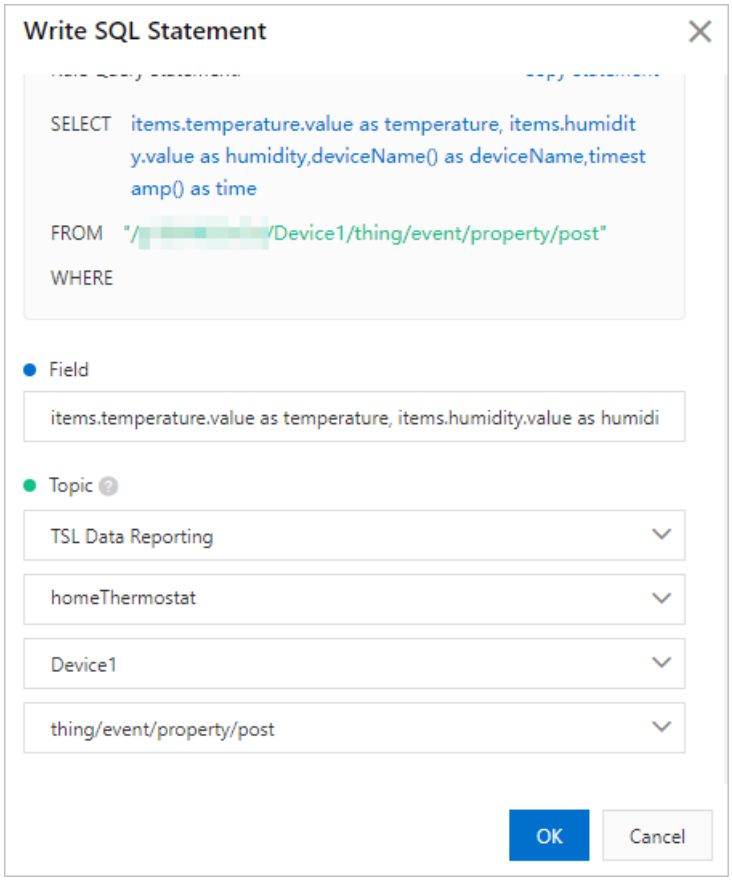
iv. After you create the product, you can go to the **Product Details** page of the product and configure the product based on your business requirements. For example, you can create a topic category or define a Thing Specification Language (TSL) model. For more information, see What is a topic? or What is a TSL model?

v. In the left-side navigation pane, choose **Devices > Devices**. On the Devices page, click Add Device and configure the parameters. For more information, see Create a device.

vi. In the left-side navigation pane, choose **Rules > Data Forwarding**. On the Data Forwarding page, click Create Rule and configure the parameters.

vii. On the Data Forwarding page, find the rule and click **View** in the Actions column.

viii. In the **Data Processing** section of the **Data Forwarding Rule** page, click **Write SQL**. In the Write SQL dialog box, write SQL statements for the rule. Then, debug the SQL statements. For more information, see SQL statements.



ix. In the **Data Forwarding** section, click **Add Operation**. In the Add Operation dialog box, configure the parameters. The operation is used to forward device data to a topic in DataHub. For more information, see Configure a data forwarding rule.

x. After the configuration is complete, go to the **Data Forwarding** page. Find the rule and click **Start** in the Actions column to enable the rule.

After you enable the rule, use a simulated device to send a message and check whether the message can be forwarded to DataHub. On the **Device Log** page, you can view the logs of the simulated device. On the DataHub console, you can view the changes in the data size of the shards of the topic to which device data is forwarded. You can also use the data sampling feature to view the message content in the shards.

3. Configure a device and connect the device to IoT Platform.

   This example shows how to use Link SDK for Java to configure the device. Click Demo SDK for Java to download the sample code.

   After you develop the device SDK, install the device SDK on the physical device. After you power on the device and connect the device to IoT Platform, the device can report data to IoT Platform. IoT Platform forwards the data to DataHub by using the rules engine. After Realtime Compute for Apache Flink processes the data, the processed data is written to ApsaraDB RDS for MySQL.

4. Perform a test. After the configuration is complete, the number of active devices is updated in real time on the dashboard each time a device connects to IoT Platform and sends messages.

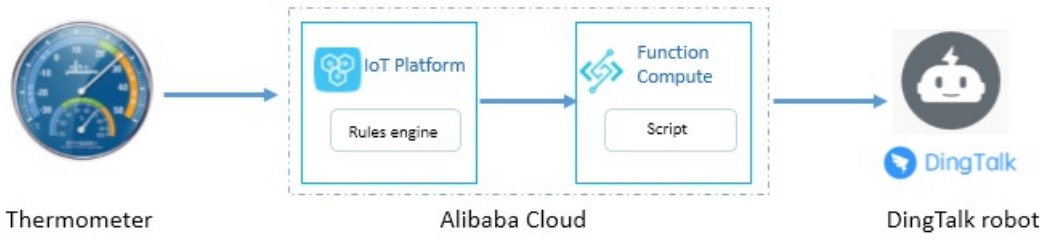
5.2. Push device data to DingTalk groups

This article describes how to push the data that is submitted by a device to a DingTalk group by using a data forwarding rule. A thermo-hygrometer is used in this example.

Scenario

Push the data that is submitted by thermo-hygrometers in office rooms to DingTalk chatbots.

Process



Step 1: Create a product and a device

1. Log on to the IoT Platform console.
2. In the left-side navigation pane, choose **Devices > Products** and create a thermo-hygrometer product. When you create the product, set Node Type to Directly Connected Device.

   Use default values for other parameters. For more information, see Create a product.
3. Click **Create TSL**. On the **Define Feature** tab, click **Edit Draft**. In the **Default Module** section, add custom features to the product.

   In this example, add the temperature and humidity properties to the product. For more information, see Add a TSL feature.
4. In the left-side navigation pane, choose **Devices > Devices** . Create a device named *TH_sensor* under the product. For more information, see Create a device.

   In the **The devices have been added** dialog box, click **Learn More** to obtain the device certificate (ProductKey, DeviceName, and DeviceSecret). You must secure the device certificate. The certificate is a key credential for subsequent communication between the device and IoT Platform.
5. On the **Devices** tab, find the device and click **View** in the Actions column. The **Device Details** page appears. In the **Tag Information** section, click **Edit** to add tags to the device.

   In this example, add the following two tags. For more information, see Tags.

Key	Value	Description
tag	Room 00XS, Floor F, Building X, YY Town	The location of the device.
deviceISN	T20180102X	The serial number (SN) of the device.

Step 2: Configure Function Compute

Function Compute is an event-driven, fully managed computing service. The programming languages that are supported by Function Compute include Java, Node.js, and Python.For more information, see How to use Function Compute.

1. Specify the webhook URL of a DingTalk chatbot.
   i. Log on to DingTalk on your PC.
   ii. In the DingTalk chat window, click the icon. On the panel that appears, click **Group Assistant**.
   iii. Click **Add Robot**, and then click the icon.
   iv. Click **Custom**, and then click **Add**.
   v. Specify the Chatbot name and Security Settings parameters, select **I have read and accepted《DingTalk Custom Robot Service Terms of Service》**, and then click **Finished**.
   vi. Click **Copy** to save the webhook URL on your PC.
2. Write a Function Compute script.

   In this example, the Node.js runtime environment is used. The function obtains device data from IoT Platform, processes the data based on the specified DingTalk message format, and then sends the data the webhook URL of the specified DingTalk chatbot by using the HTTPS POST method. The device data includes the device location, device SN, real-time temperature and humidity data, and the time when the device submits the data to IoT Platform.

   After you write the script, name the script file as *index.js*, and then compress it into the *index.zip* file. The following sample code can be used.

   You must replace accessToken with the value of the access_token parameter in the webhook URL.

```
const https = require('https');
const accessToken = 'Specify the value of the access_token parameter in the webhook URL
.
module.exports.handler = function(event, context, callback) {
var eventJson = JSON.parse(event.toString());
// DingTalk message format
const postData = JSON.stringify({
"msgtype": "markdown",
"markdown": {
"title": "Thermo-hygrometer",
"text": "#### Submit data\n" +
"> Device location: " + eventJson.tag + "\n\n" +
"> Device SN: " + eventJson.isn+ "\n\n" +
"> Temperature: " + eventJson.temperature + "℃\n\n" +
"> Humidity: " + eventJson.humidity + "%\n\n" +
"> ###### " + eventJson.time + " published by [IoT Platform](https://www.aliyun.com/pro
duct/iot) \n"
},
"at": {
"isAtAll": false
}
});
const options = {
hostname: 'oapi.dingtalk.com',
port: 443,
path: '/robot/send?access_token=' + accessToken,
method: 'POST',
headers: {
'Content-Type': 'application/json',
'Content-Length': Buffer.byteLength(postData)
}
};
const req = https.request(options, (res) => {
res.setEncoding('utf8');
res.on('data', (chunk) => {});
res.on('end', () => {
callback(null, 'success');
});
});
// Return an error.
req.on('error', (e) => {
callback(e);
});
// Write the data.
req.write(postData);
req.end();
};
```

3. Create a service and a function.
   i. For information about how to activate Alibaba Cloud Function Compute, see Activate Function Compute.
   ii. Log on to the Function Compute console. In the left-side navigation pane, click **Services and Functions**.

iii. Click **Create Service**, set Service Name to *IoT_Service*, and then click **Submit**.

iv. On the **Services** tab, find **IoT_Service** and click **Create Function**.

v. Move the pointer over **Event Function** and click **Configure and Deploy**.

vi. Set the parameters of the function. Use the default values for other parameters, and then click **Create**.

Parameter	Description
Function Name	Enter *pushData2DingTalk*.
Runtime	Select **Node.JS 6.x** and set **Upload Code** to **Upload Zip File**. Click **Upload Code** next to Upload Zip File and upload the *index.zip* file that is created in Step 2.

Step 3: Forward data to Function Compute

Configure a data forwarding rule to forward the temperature and humidity data submitted by the TH_sensor device to the **pushData2DingTalk** function.

1. Log on to the IoT Platform console.In the left-side navigation pane, choose **Rules > Data Forwarding** . On the Data Forwarding page, click **Create Rule**. Set the parameters and click **OK**.
2. On the **Data Forwarding Rule** page, click **Write SQL** to edit an SQL statement.

   In this example, specify the following fields to filter data:

   - The fields that indicate the device information: deviceName, tag, and deviceISN.
   - The fields that indicate the submitted payloads: temperature, and humidity.

   The following script shows the sample SQL statement:

   ```
   SELECT
   deviceName() as deviceName,
   attribute('tag') as tag, attribute('deviceISN') as isn,
   items.temperature.value as temperature, items.humidity.value as humidity,
   timestamp('yyyy-MM-dd HH:mm:ss') as time
   FROM
   "/g5j3o***/TH_sensorthing/event/property/post"
   ```

3. On the **Data Forwarding Rule** page, click **Add Operation** to forward data to Function Compute.

   In this example, select the created **IoT_Service** service and **pushData2DingTalk** function. For more information, see Forward data to Function Compute.

4. On the Data Forwarding page, find the rule and click **Start** in the Actions column to enable the rule.

Step 4: Connect the thermo-hygrometer device to IoT Platform and submit temperature and humidity data

Connect the device to IoT Platform over MQTT by using the device certificate. Then, simulate the submission of temperature and humidity data.

1. Download and install Node.js on the Windows or Linux operating system. In this example, the Windows operating system is used.

   ```
   node --version
   ```

2. Create a JavaScript file such as iot_device.js on your PC to store the Node.js sample code.

The following Node.js sample code can be used:

```
const mqtt = require('aliyun-iot-mqtt');
// 1. Specify the device identity information
var options = {
    productKey: "a1***",
    deviceName: "TH_sensor",
    deviceSecret: "9JtvE***",
    regionId: 'cn-shanghai'
};
// 1. Create a connection
const client = mqtt.getAliyunIotMqttClient(options);
// 2. Configure a listener to receive downstream commands
client.subscribe(`/${options.productKey}/${options.deviceName}/user/get`)
client.on('message', function(topic, message) {
    console.log("topic " + topic)
    console.log("message " + message)
})
setInterval(function() {
    // 3.Submit data
    client.publish(`/sys/${options.productKey}/${options.deviceName}/thing/event/proper
ty/post`, getPostData(), { qos: 0 });
}, 6 * 1000);
function getPostData() {
    const payloadJson = {
        id: Date.now(),
        version: "1.0",
        params: {
            Temperature: Math.floor((Math.random() * 20) + 10),
            Humidity: Math.floor((Math.random() * 20) + 10)
        },
        method: "thing.event.property.post"
    }
    console.log("payloadJson " + JSON.stringify(payloadJson))
    return JSON.stringify(payloadJson);
}
```

Parameter	Example	Description
productKey	a1***	The ProductKey of the product to which the device belongs. You can view the ProductKey on the Device Details page of the IoT Platform console.
deviceName	TH_sensor	The DeviceName of the device. You can view the DeviceName on the Device Details page of the IoT Platform console.
deviceSecret	9JtvE***	The DeviceSecret of the device. You can view the DeviceSecret on the Device Details page of the IoT Platform console.

Parameter	Example	Description
regionId	cn-shanghai	The ID of the region where the MQTT device resides. For more information about region IDs, see Regions and zones.

3. Open the command-line interface and run the `cd` command to go to the directory where the iot_device.js file resides. In this directory, run the NPM command to download the aliyun-iot-mqtt library.

```
npm install aliyun-iot-mqtt -S
```

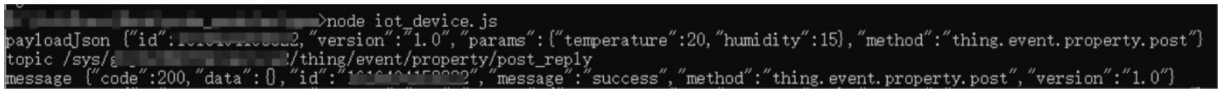
The following figure shows the downloaded library file.

4. In the command-line interface, enter the following command to run the iot_device.js file. This way, the device is enabled and data is submitted.

```
node iot_device.js
```

Result

The following figure shows the result.



The following figure shows the message that is received by the DingTalk chatbot.