

---

# InGateway Documentation

*Release 0.0.1*

zhangning

May 06, 2023



<b>1</b>	<b>InGateway Documentation Site Navigation</b>	<b>1</b>
1.1	Device Supervisor App User Manual . . . . .	1
1.2	AWS IoT User Manual . . . . .	92
1.3	Azure IoT User Manual . . . . .	116
1.4	DeviceSupervisor 2.0 Upgrade Notes . . . . .	145





---

## InGateway Documentation Site Navigation

---

The Device Supervisor App provides users with convenient and reliable data collection, data processing, and data upload to cloud functions. It supports ISO on TCP, ModbusRTU and other industrial protocol analysis. If you want to quickly realize the data collection of multiple industrial protocols and preprocess the device data locally, the filtered data can be uploaded to the self-built MQTT cloud platform after simple configuration, then Device Supervisor will be your ideal choice.

### 1.1 Device Supervisor App User Manual

Device Supervisor App (Device Supervisor) allows users to collect and process data and upload data to the cloud conveniently and supports data parsing over multiple industrial protocols such as ISO on TCP and ModbusRTU. This manual takes collecting PLC data and uploading it to the ThingsBoard cloud platform as an example to describe how to collect PLC data and upload it to the cloud through Device Supervisor App. Hereinafter, InGateway501 is referred to as **IG501**, InGateway502 is referred to as **IG502** and InGateway902 is referred to as **IG902**.

- *Overview*
- *1. Prepare the hardware and data collection environment*
  - *1.1 Hardware wiring*
    - \* *1.1.1 Ethernet wiring*
    - \* *1.1.2 Serial port wiring*
  - *1.2 Configure InGateway to access PLC*

- *1.3 Configure InGateway to connect Internet*
  - *1.4 Update the InGateway software version*
- *2. Configure data collection for Device Supervisor*
  - *2.1 Install and run Device Supervisor*
  - *2.2 Configure data collection*
    - \* *2.2.1 Add a PLC*
    - \* *2.2.2 Add a variable*
    - \* *2.2.3 Configure an alarm policy*
    - \* *2.2.4 Configure a group*
- *3. Report and monitor the PLC data*
  - *3.1 Monitor the PLC data locally*
    - \* *3.1.1 Monitor data collection locally*
    - \* *3.1.2 Monitor the alarm locally*
  - *3.2 Monitor the PLC data on cloud*
    - \* *3.2.1 Configure ThingsBoard*
    - \* *3.2.2 Configure a cloud service to report and receive data*
- *Appendix*
  - *Importing/exporting data collection configuration*
  - *Messages Management (custom MQTT publish/subscribe)*
    - \* *Configure Publish Messages*
    - \* *Configure Subscribe Messages*
    - \* *Device Supervisor API Description*
    - \* *Device Supervisor API Callback Function Description*
  - *Parameter settings*
  - *Gateway other configuration*
  - *ThingsBoard reference flowchart*
    - \* *Add devices and assets*
    - \* *Transmit the PLC data to ThingsBoard devices*
    - \* *Configure a visual dashboard*
- *FAQ*

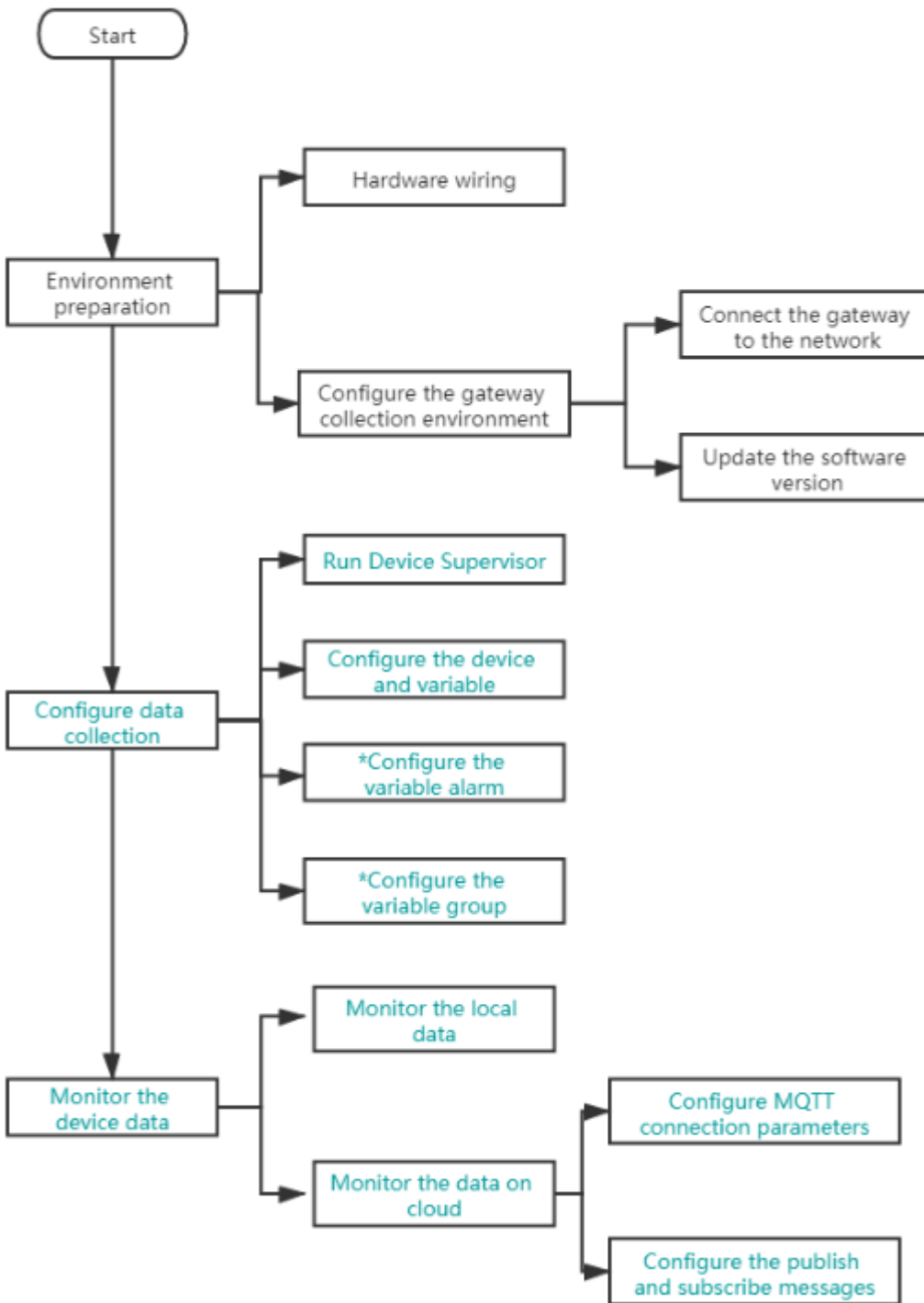
- *Check whether the cloud service script is correct*
- *Check whether the App's cloud service output is correct*

### 1.1.1 Overview

Prepare the following items:

- Edge computing gateway IG501/IG502/IG902
- PLC
- Network cable/serial port cable
- \*Firmware, SDKs, and apps required for software version update
  - Firmware version: V2.0.0.r12622 or later
  - SDK version: py3sdk-V1.3.7 or later
  - App version: 1.1.2 or later
- \*ThingsBoard demo account

The whole process is shown in the figure below:



### 1.1.2 1. Prepare the hardware and data collection environment

- *1.1 Hardware wiring*
- *1.2 Configure InGateway to access PLC*
- *1.3 Configure InGateway to connect Internet*

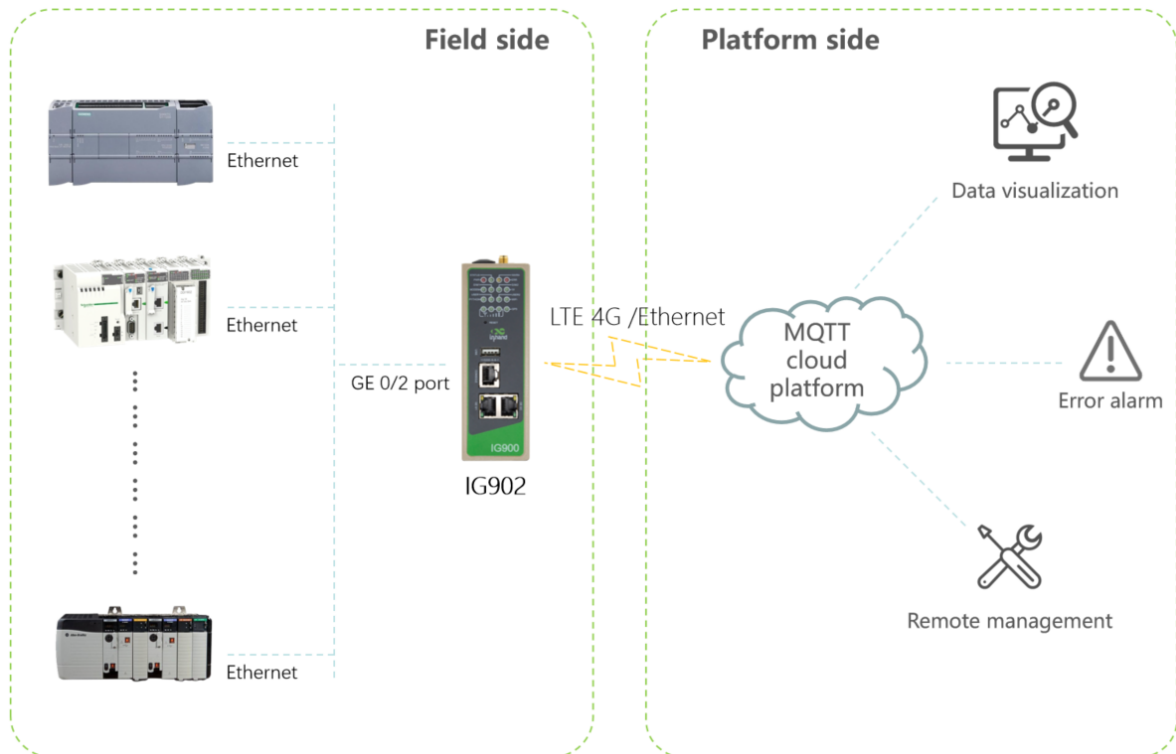
- 1.4 Update the InGateway software version

## 1.1 Hardware wiring

### 1.1.1 Ethernet wiring

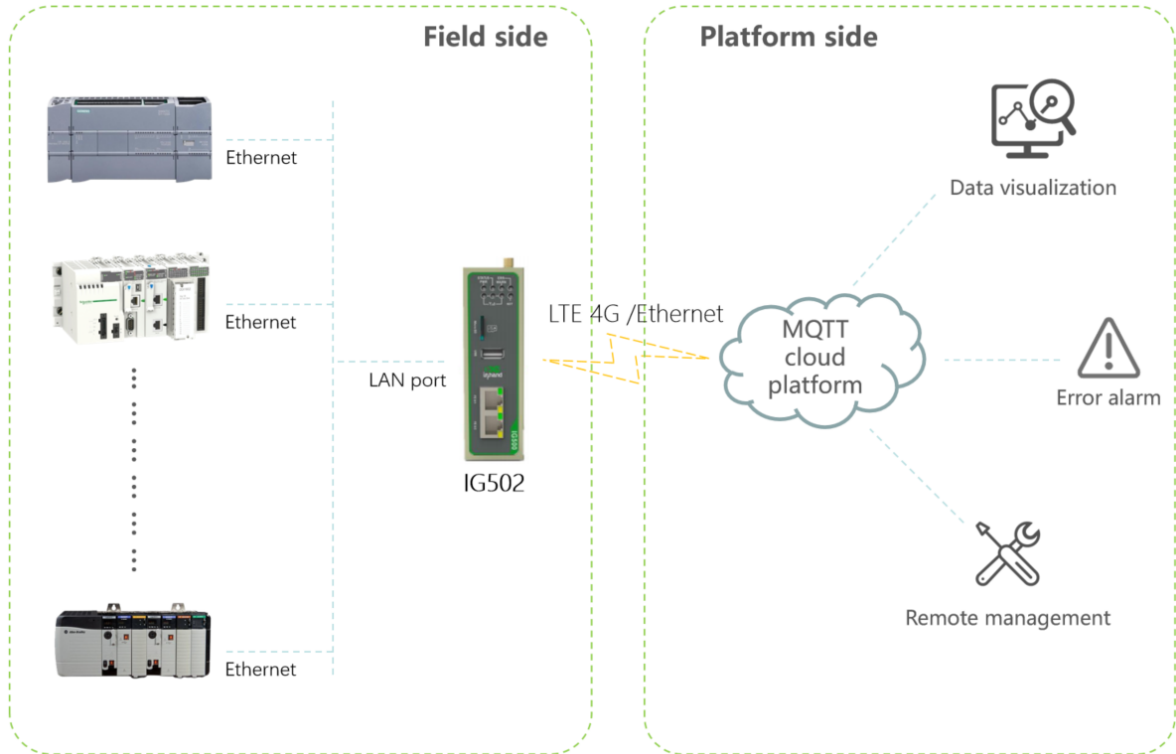
- Ethernet wiring for IG902

Power on IG902 and connect IG902 and the PLC through an Ethernet cable according to the topology.



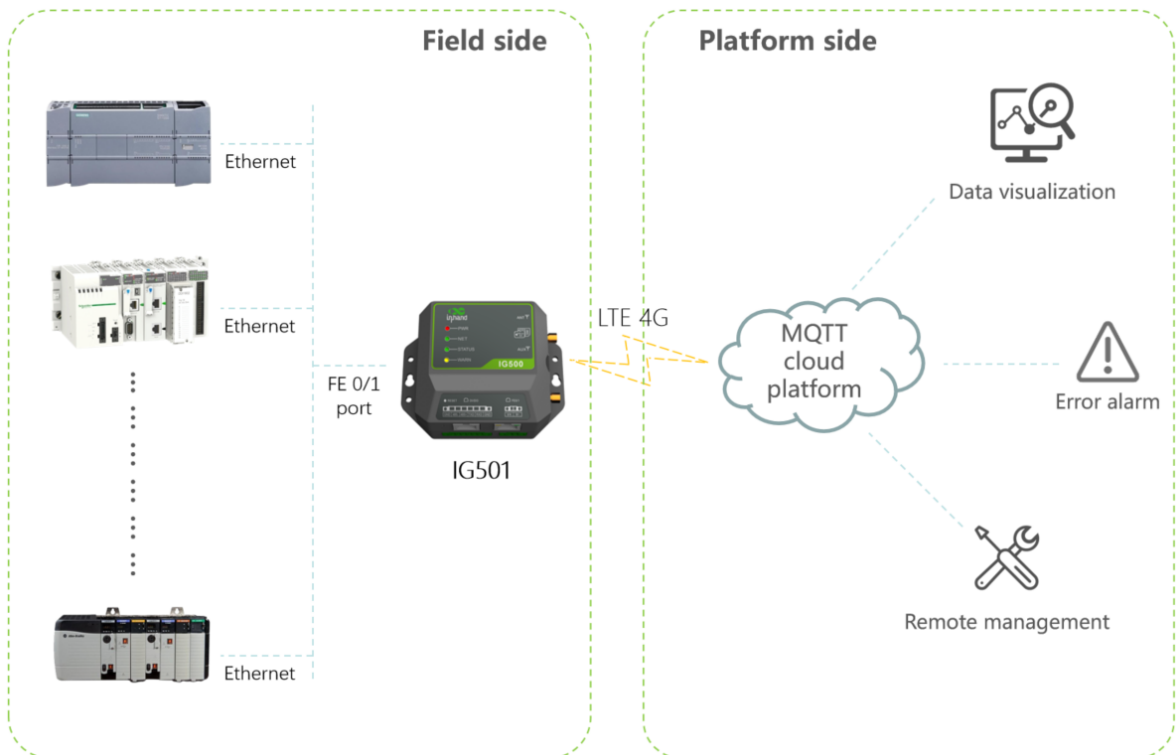
- Ethernet wiring for IG502

Power on IG502 and connect IG502 and the PLC through an Ethernet cable according to the topology.



- Ethernet wiring for IG501

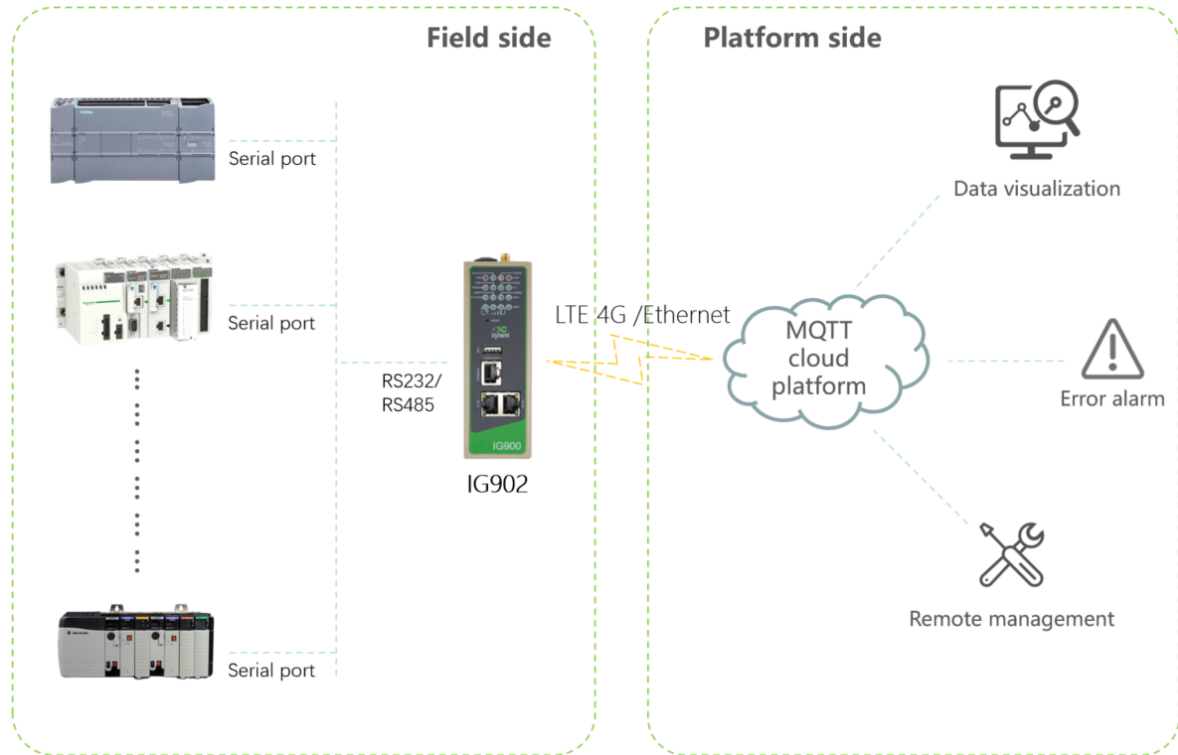
Power on IG501 and connect IG501 and the PLC through an Ethernet cable according to the topology.



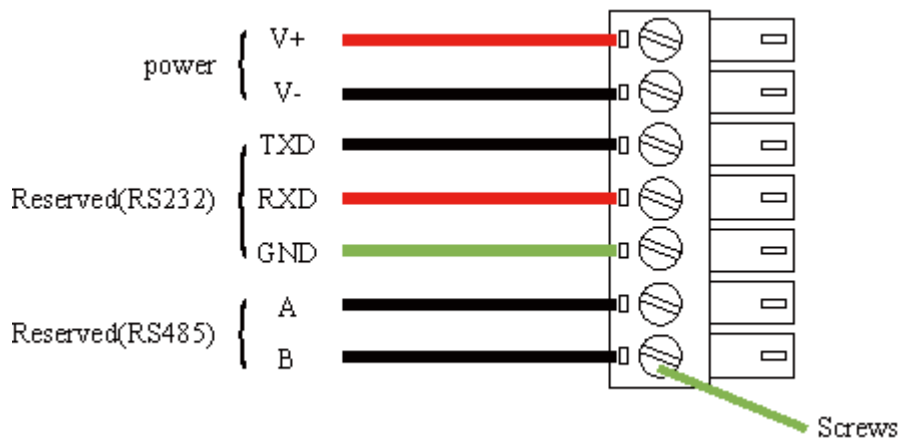
### 1.1.2 Serial port wiring

- Serial port wiring for IG902

Power on IG902 and connect IG902 and the PLC according to the topology.

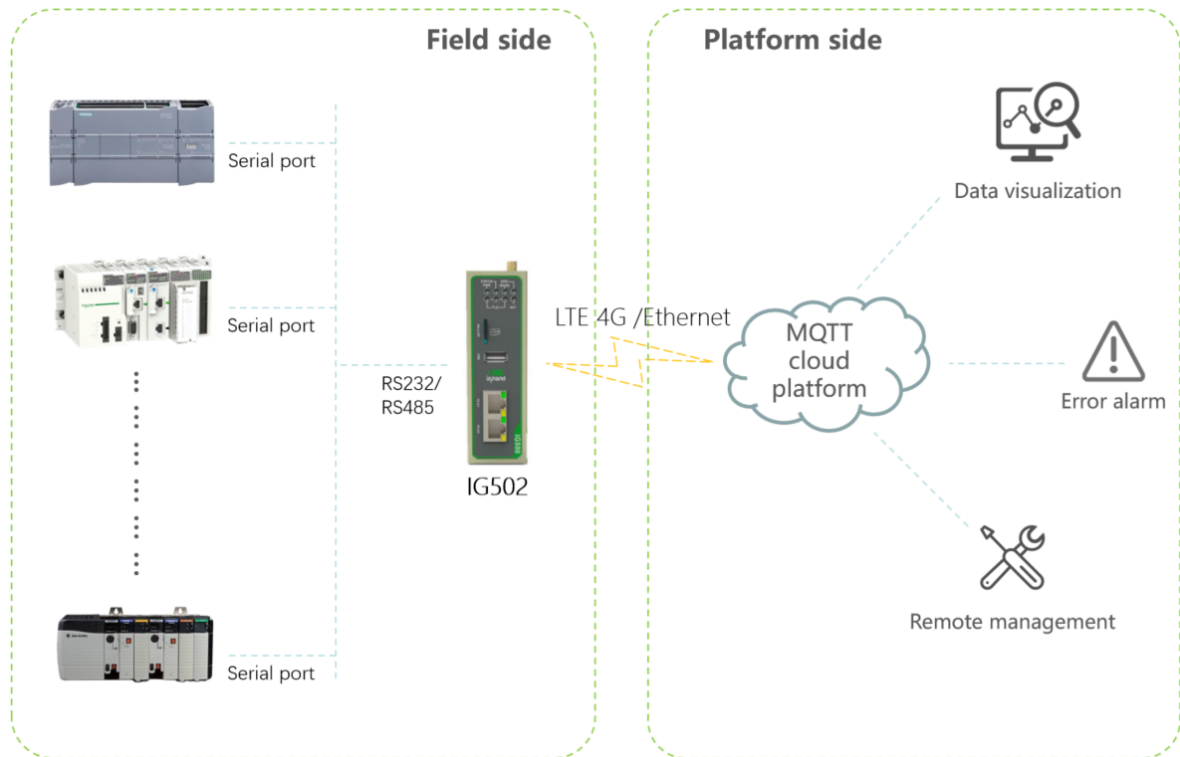


The following figure describes how to connect serial port terminals of IG902:

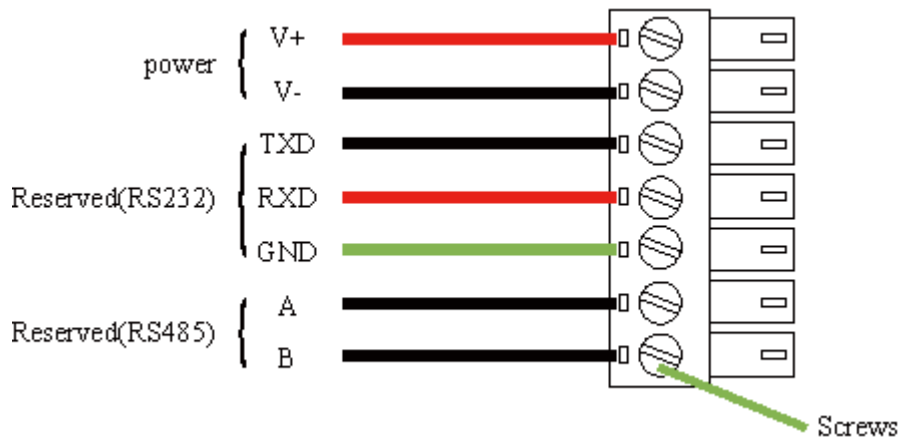


- Serial port wiring for IG502

Power on IG502 and connect IG502 and the PLC according to the topology.



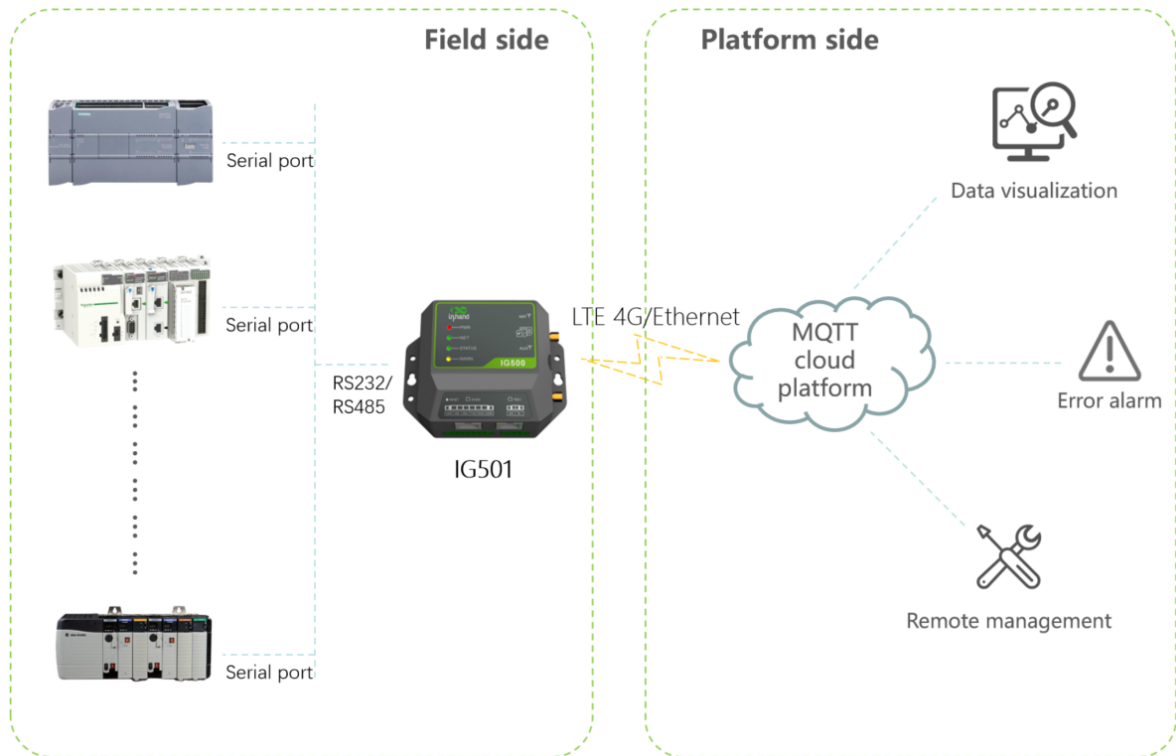
The following figure describes how to connect serial port terminals of IG502:



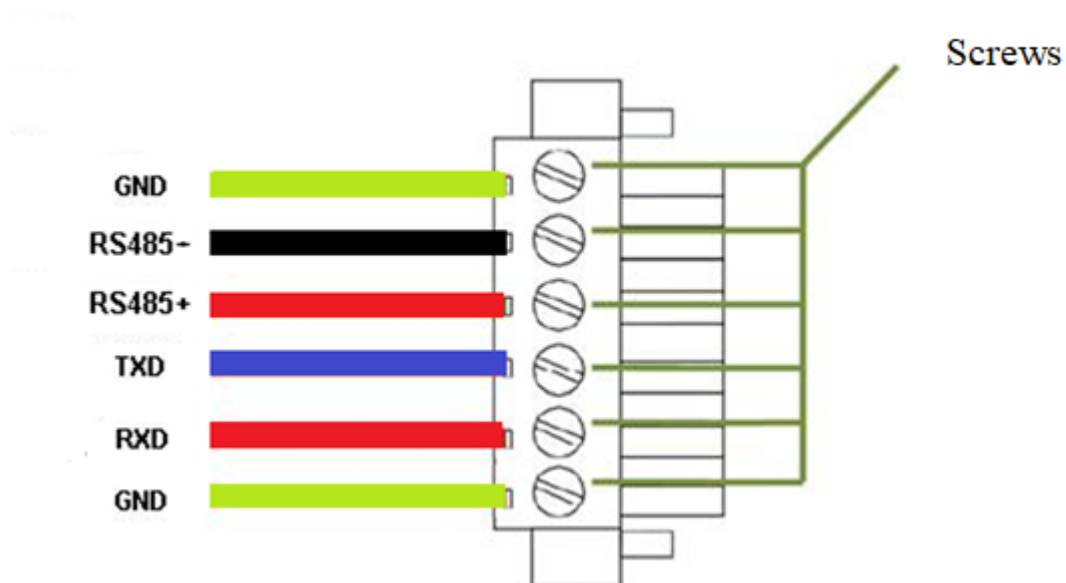
- Serial port wiring for IG501

Power on IG501 and connect IG501 and the PLC according to the topology.





The following figure describes how to connect serial port terminals of IG501:



## 1.2 Configure InGateway to access PLC

- The default IP address of IG902's GE 0/2 port is 192.168.2.1. To enable IG902 to access the Ethernet-based PLC over the GE 0/2 port, you need to set the GE 0/2 port to be in the same network

segment of the PLC. For more information about the setting method, see [Access the IG902](#).

- The default IP address of IG502' s LAN port is 192.168.2.1. To enable IG502 to access the Ethernet-based PLC over the LAN port, you need to set the LAN port to be in the same network segment of the PLC. For more information about the setting method, see [Access the IG502](#).
- The default IP address of IG501' s FE 0/1 port is 192.168.1.1. To enable IG501 to access the Ethernet-based PLC over the FE 0/1 port, you need to set the FE 0/1 port to be in the same network segment of the PLC. For more information about the setting method, see [Access the IG501](#).

### 1.3 Configure InGateway to connect Internet

- Configure the IG902 to connect Internet by referring to [Connect IG902 to the Internet](#).
- Configure the IG502 to connect Internet by referring to [Connect IG502 to the Internet](#).
- Configure the IG501 to connect Internet by referring to [Connect IG501 to the Internet](#).

### 1.4 Update the InGateway software version

If you want to get the latest InGateway and its functional characteristics, please visit the [Resource](#). To update the software version, see the following links:

- [Update the IG902 software version](#) To use Device Supervisor, IG902' s firmware version must be V2.0.0.r12537 or later, and the SDK version must be py3sdk-V1.4.2 or later.
- [Update the IG502 software version](#) To use Device Supervisor, IG502' s firmware version must be V2.0.0.r13595 or later, and the SDK version must be py3sdk-V1.4.2 or later.
- [Update the IG501 software version](#) To use Device Supervisor, IG501' s firmware version must be V2.0.0.r12884 or later, and the SDK version must be py3sdk-V1.4.0 or later.

## 1.1.3 2. Configure data collection for Device Supervisor

- *2.1 Install and run Device Supervisor*
- *2.2 Configure data collection*

### 2.1 Install and run Device Supervisor

- Install and run Python apps on IG902 by referring to [Install and run Python apps on IG902](#). To download Device Supervisor, please visit the [Resource](#). After Device Supervisor runs normally, the following figure is displayed:

Overview / Edge Computing / Python Edge Computing

### Python Engine

SDK Version: 1.3.8 [Upgrade](#) Enable Debug Mode: ☒

Python Version: Python3 Username: pyuser

Used User Storage: 402MB/6GB 6% Password: 4DqrGQGB+c5\_

### APP

App Status Entire Operation [▶](#) [⏸](#) [↺](#)

App Name	App Version	SDK Version	State	Uptime	Log	Operation
device_supervisor	1.1.3	1.3.7	<b>RUNNING</b>	00:04:35	<a href="#">↓</a> <a href="#">🗑</a> <a href="#">🔍</a>	<a href="#">⏸</a> <a href="#">↺</a>

App List

Enable	App Name	App Version	SDK Version	Start Parameters	Operation <a href="#">+</a>
<input checked="" type="checkbox"/>	device_supervisor	1.1.3	1.3.7		<a href="#">🗑</a>

[Submit](#) [Reset](#)

- Install and run Python apps on IG502 by referring to [Install and run Python apps on IG502](#). To download Device Supervisor, please visit the [Resource](#). After Device Supervisor runs normally, the following figure is displayed:

Overview / Edge Computing / Python Edge Computing




### Python Engine






SDK Version: 1.3.8 [Upgrade](#) Enable Debug Mode: ☒

Python Version: Python3 Username: pyuser



Used User Storage: 402MB/6GB 6% Password: 4DqrGQGB+c5\_

### APP

App Status Entire Operation   

App Name	App Version	SDK Version	State	Uptime	Log	Operation
device_supervisor	1.1.3	1.3.7	<b>RUNNING</b>	00:04:35	  	 

App List

Enable	App Name	App Version	SDK Version	Start Parameters	Operation 
<input checked="" type="checkbox"/>	device_supervisor	1.1.3	1.3.7		

[Submit](#) [Reset](#)

- Install and run Python apps on IG501 by referring to [Install and run Python apps on IG501](#). To download Device Supervisor, please visit the [Resource](#). After Device Supervisor runs normally, the following figure is displayed:

Overview / Edge Computing / Python Edge Computing

### Python Engine

Python Engine ☒

SDK Version: 1.3.8 [Upgrade](#)

Python Version: Python3

Used User Storage: 402MB/6GB 6%

Enable Debug Mode: ☒

Username: pyuser

Password: 4DqrGQGB+c5\_

### APP

App Status

Entire Operation [▶](#) [⏸](#) [🔄](#)

App Name	App Version	SDK Version	State	Uptime	Log	Operation
device_supervisor	1.1.3	1.3.7	<b>RUNNING</b>	00:04:35	<a href="#">↓</a> <a href="#">🗑</a> <a href="#">🔍</a>	<a href="#">⏸</a> <a href="#">🔄</a>

App List

Enable	App Name	App Version	SDK Version	Start Parameters	Operation <a href="#">+</a>
<input checked="" type="checkbox"/>	device_supervisor	1.1.3	1.3.7		<a href="#">🗑</a>

[Submit](#) [Reset](#)

## 2.2 Configure data collection

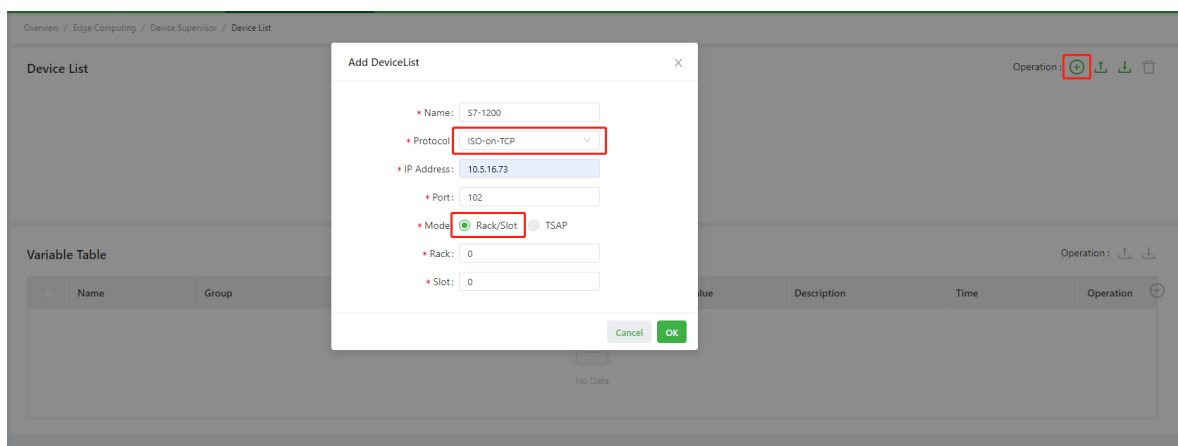
- *2.2.1 Add a PLC*
- *2.2.2 Add a variable*
- *2.2.3 Configure an alarm policy*
- *2.2.4 Configure a group*

### 2.2.1 Add a PLC

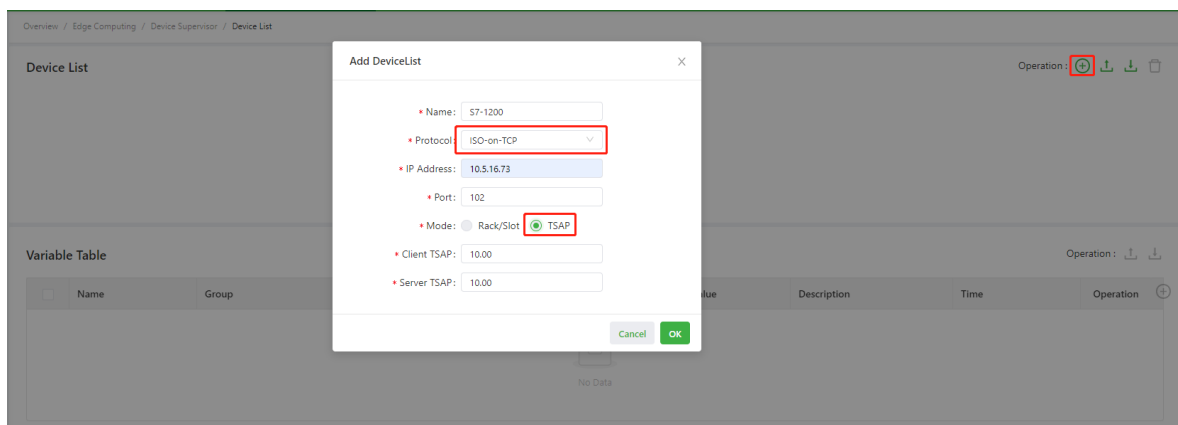
- Add a PLC that communicates over ISO on TCP

Choose **Edge Computing > Device Supervisor > Device List**, and click **Add PLC**. On the device adding page, select **ISO on TCP** as the PLC protocol and configure the PLC communication parameters. Note: The device name must be unique.

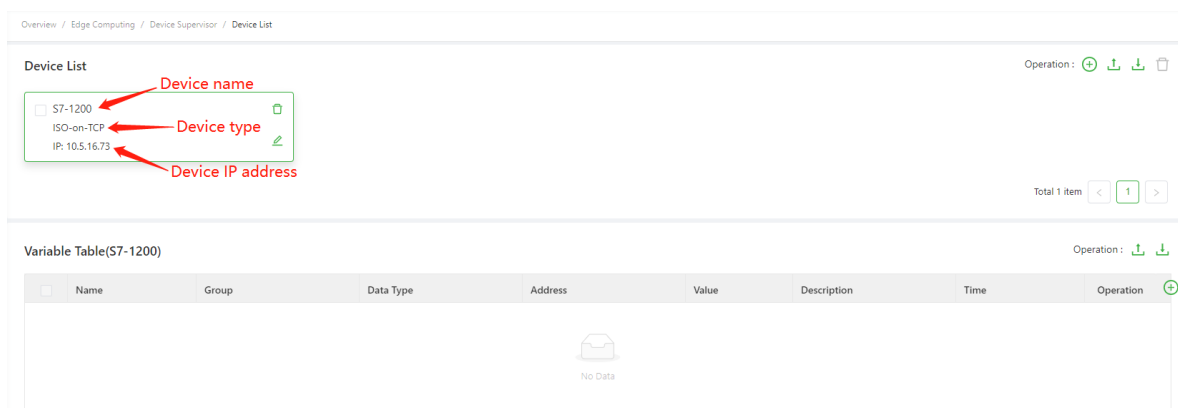
The following are examples of adding PLCs S7-1500, S7-1200, S7-400, and S7-300(the mode is Rack/Slot). Configure the rack number and slot number to 0 and 1 respectively.



The following are examples of adding S7-200, S7-200 Smart and Siemens LOGO series PLCs (the mode is TSAP). Note: When adding S7-200 Smart, the Client TSAP configuration is 02.00, and the Server TSAP configuration is 02.01; the rest of the series are configured according to the actual situation.



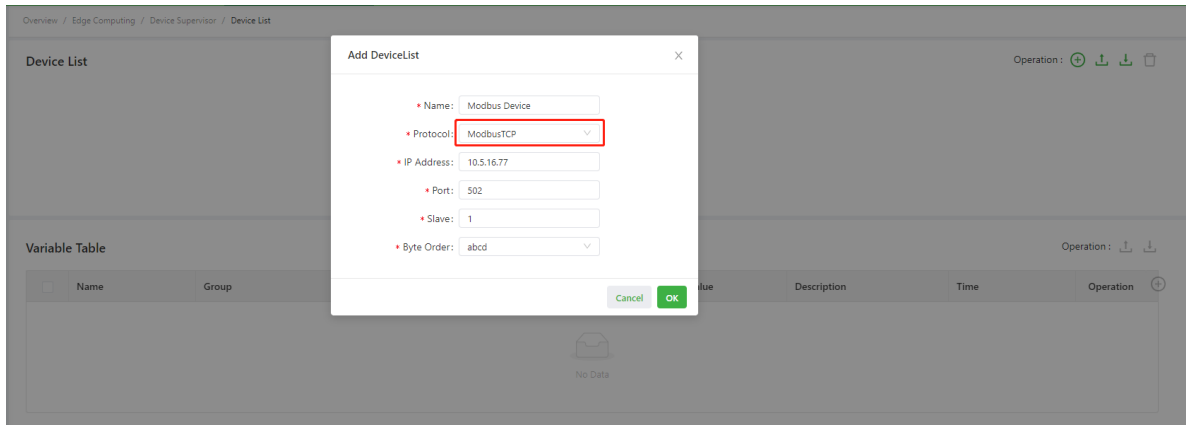
After the PLC is added, the page is as follows:



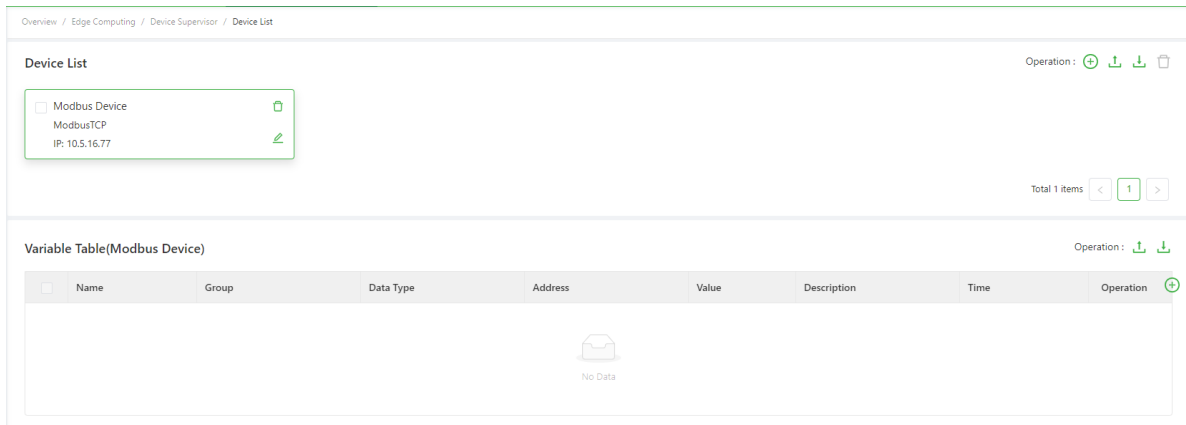
- Add a PLC that communicates over ModbusTCP

Choose **Edge Computing > Device Supervisor > Device List**, and click **Add PLC**. On the device adding page, select **ModbusTCP** as the PLC protocol and configure the PLC communication parameters. (The default port number and byte order are 502 and abcd respectively. Adjust them as

needed.) Note: The device name must be unique.

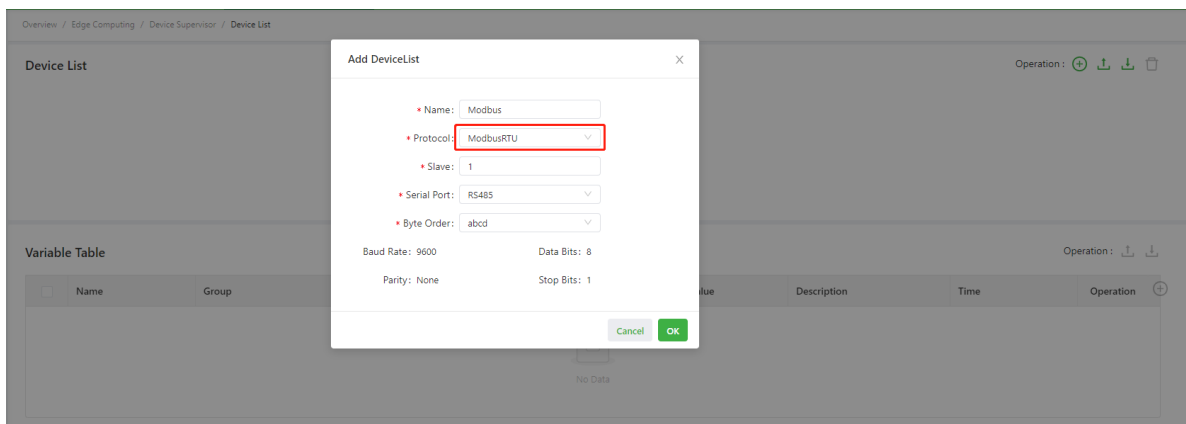


After the PLC is added, the page is as follows:







- Add a PLC that communicates over ModbusRTU

Choose **Edge Computing > Device Supervisor > Device List**, and click **Add PLC**. On the device adding page, select ModbusRTU as the PLC protocol and configure the PLC communication parameters. Note: The device name must be unique.





After the PLC is added, the page is as follows:



Overview / Edge Computing / Device Supervisor / Device List

**Device List** Operation:    

☐ Modbus  
 ModbusRTU  
 Slave: 1

Total 1 items < 1 >

**Variable Table(Modbus)** Operation:  


<input type="checkbox"/>	Name	Group	Data Type	Address	Value	Description	Time	Operation 
 No Data								


To modify communication parameters for the RS232/RS485 serial port, choose **Edge Computing > Device Supervisor > Parameter Settings**, and modify them on the page. After modification, communication parameters of all serial ports are automatically updated to the modified ones.


Overview / Edge Computing / Device Supervisor / Parameter Settings


**Serial Port Settings**

**RS-485 Serial Port**


\* Baud Rate: 9600 


\* Data Bits: 8 


\* Parity: None 


\* Stop Bits: 1 

**RS-232 Serial Port**


\* Baud Rate: 9600 

\* Data Bits: 8 

\* Parity: None 

\* Stop Bits: 1 


**Default Parameter**

\* Log level: DEBUG 

\* Historical alarm max: 2000 (1-10000)

\* Historical data max: 100000 (1-100000)

**Custom Parameter**

Parameters	Parameters Value	Operation 
------------	------------------	---

- Add EtherNET/IP device (**App version 1.2.5 or above is required**)

Choose **Edge Computing > Device Monitoring > Device List**, and click on **Add PLC**. Select **EtherNET/IP** as the PLC protocol on the **Add Device** page, and then configure the communication parameters of the PLC. Note: The device name must be unique.



**Add to DeviceList** [X]

\* Name:

\* Protocol:

\* IP Address:

\* Port:

### 2.2.2 Add a variable

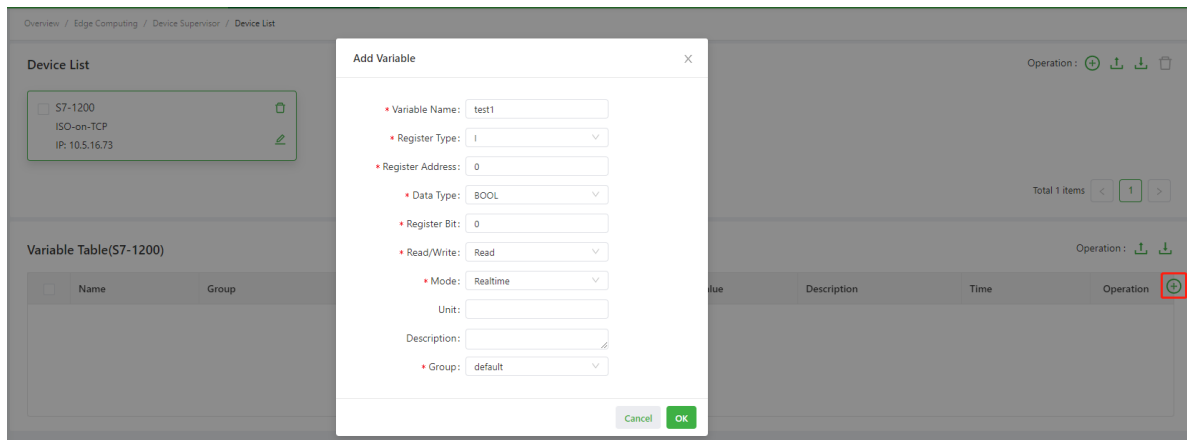
- Add the ISO on TCP variable

On the **Device List** page, click **Add** variable, and configure the variable parameters in the pop-up box.

- **Variable Name:** The variable name. (The variable name must be unique in the same device.)
- **Register Type:** The variable register type. Four types are supported: I/Q/M/DB.
- **DB Number:** The variable DB number when the register type is DB.
- **Register Address:** The variable register address.
- **Data Type:** The variable data type, including:
  - \* **BOOL:** True or False.
  - \* **BIT:** 0 or 1.
  - \* **BYTE:** An 8-bit unsigned character.
  - \* **SINT:** An 8-bit signed character.
  - \* **WORD:** A 16-bit unsigned character.
  - \* **INT:** A 16-bit signed character.
  - \* **DWORD:** A 32-bit unsigned character.
  - \* **DINT:** A 32-bit signed character.
  - \* **FLOAT:** A 32-bit floating point.

- \* **STRING:** An 8-bit string.
- \* **BCD:** A 16-bit BCD code.
- **Decimal Places:** The number of decimal places of the variable when the data type is FLOAT. The maximum value is 6.
- **Size:** The length of one read string is 1 when the data type is STRING.
- **Register Bit:** The bit offset of the variable when the data type is BOOL or BIT. Any integer from 0 to 7 is supported.
- **Read/Write:**
  - \* **Read:** Read only.
  - \* **Write:** Write only.
  - \* **Read/Write:** Read and write.
- **Mode:**
  - \* **Realtime:** Collect the variable data at the collection interval of the group it belongs and report the data at the report interval.
  - \* **Onchange:** Collect the variable data only when the data changes and report the data at the report interval.
- **Unit:** The variable unit.
- **Description:** The variable description.
- **Group:** The collection group to which the variable belongs.

The following figure is an example of adding a switch variable with the address %I0.0:



The following figure is an example of adding a byte variable with the address %IB1:

**Add Variable** [X]

\* Variable Name:

\* Register Type:

\* Register Address:

\* Data Type:

\* Read/Write:

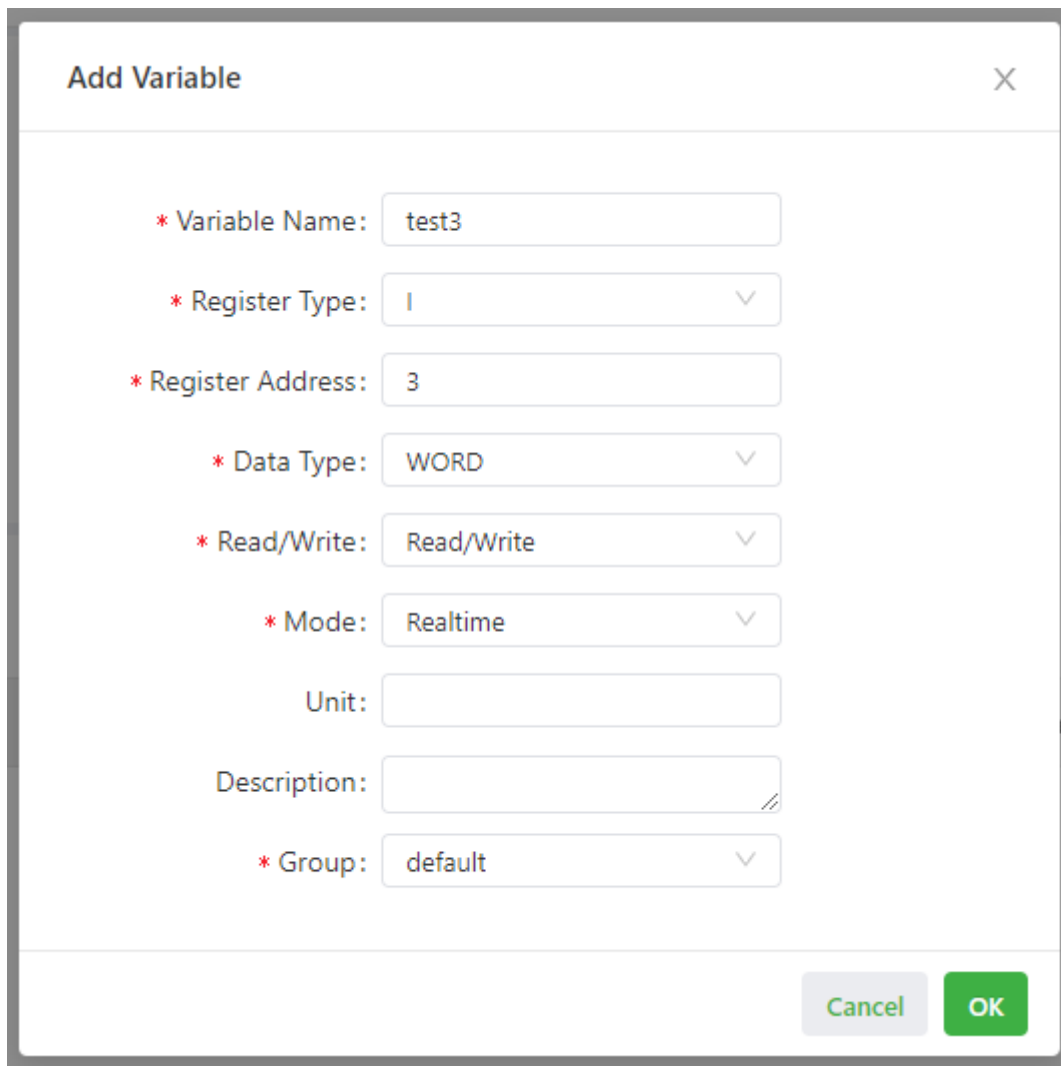
\* Mode:

Unit:

Description:

\* Group:

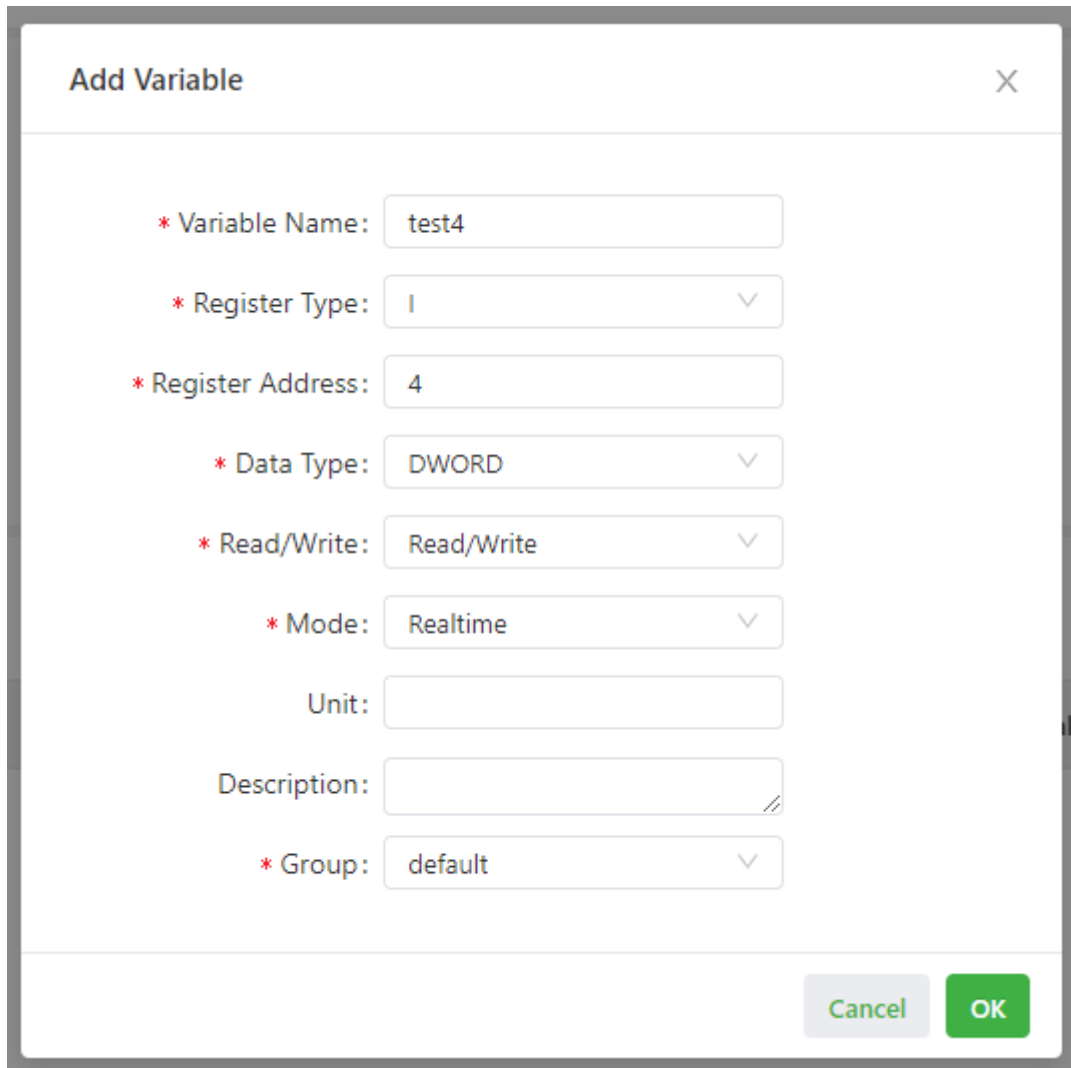
The following figure is an example of adding a word variable with the address %IW3:



The image shows a dialog box titled "Add Variable" with a close button (X) in the top right corner. The dialog contains several input fields and dropdown menus, each preceded by a red asterisk (\*). The fields are: "Variable Name" with the value "test3", "Register Type" with the value "I", "Register Address" with the value "3", "Data Type" with the value "WORD", "Read/Write" with the value "Read/Write", "Mode" with the value "Realtime", "Unit" (empty), "Description" (empty), and "Group" with the value "default". At the bottom right, there are two buttons: "Cancel" (light gray) and "OK" (green).

Field	Value
* Variable Name	test3
* Register Type	I
* Register Address	3
* Data Type	WORD
* Read/Write	Read/Write
* Mode	Realtime
Unit	
Description	
* Group	default

The following figure is an example of adding a dual-word variable with the address %ID4:



The image shows a dialog box titled "Add Variable" with a close button (X) in the top right corner. The dialog contains several input fields and dropdown menus, each preceded by a red asterisk (\*). The fields are: "Variable Name" with the text "test4"; "Register Type" with a dropdown showing "I"; "Register Address" with the text "4"; "Data Type" with a dropdown showing "DWORD"; "Read/Write" with a dropdown showing "Read/Write"; "Mode" with a dropdown showing "Realtime"; "Unit" with an empty text field; "Description" with an empty text field and a small icon in the bottom right corner; and "Group" with a dropdown showing "default". At the bottom right of the dialog are two buttons: "Cancel" (light gray) and "OK" (green).

**Add Variable** [X]

\* Variable Name: test4

\* Register Type: I

\* Register Address: 4

\* Data Type: DWORD

\* Read/Write: Read/Write

\* Mode: Realtime

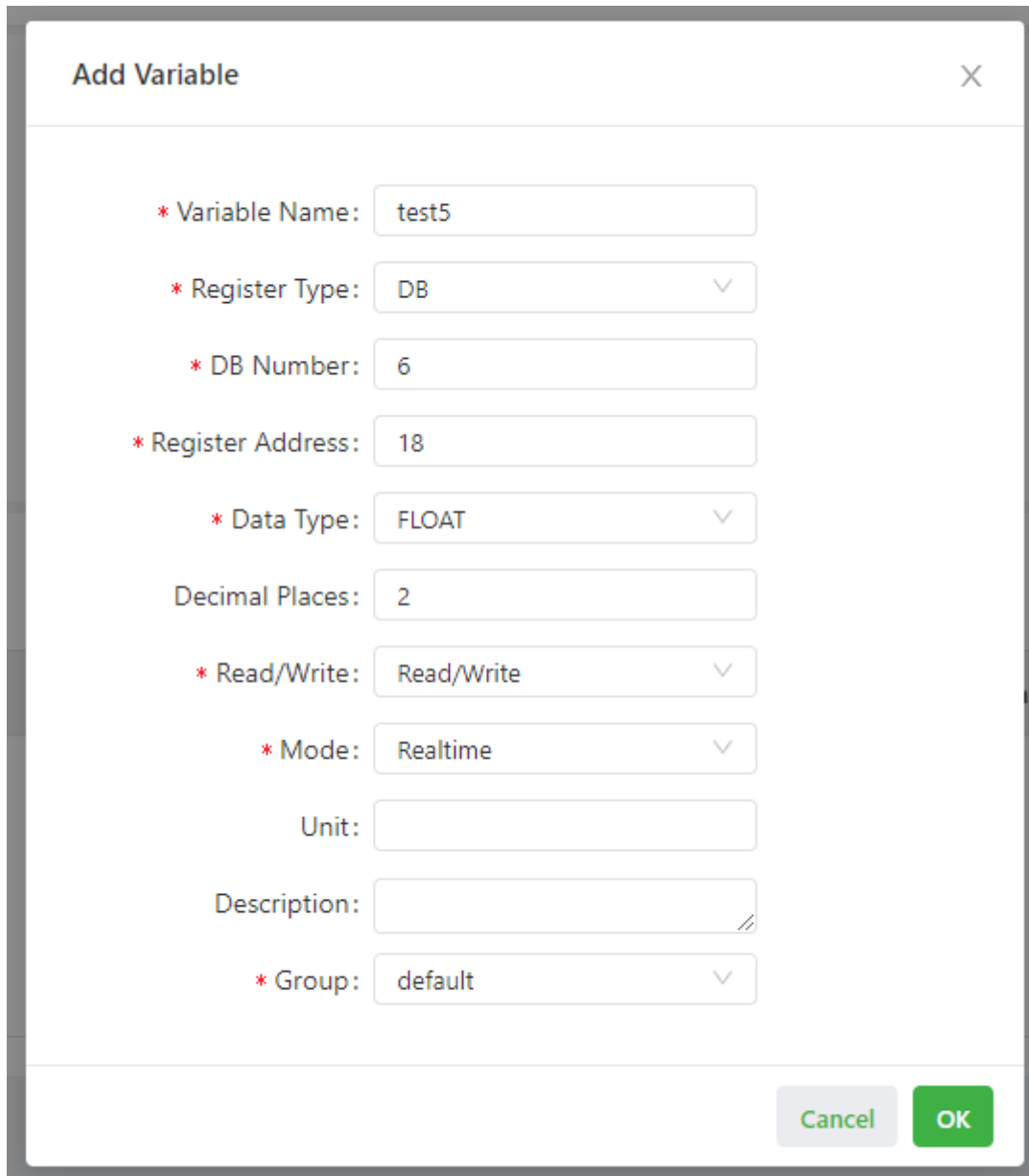
Unit:

Description:

\* Group: default

Cancel OK

The following figure is an example of adding a floating point variable with the address %DB6.DBD18:

A screenshot of a web-based dialog box titled "Add Variable" with a close button (X) in the top right corner. The dialog contains several input fields and dropdown menus for configuring a variable. The fields are: "Variable Name" (text input with "test5"), "Register Type" (dropdown with "DB"), "DB Number" (text input with "6"), "Register Address" (text input with "18"), "Data Type" (dropdown with "FLOAT"), "Decimal Places" (text input with "2"), "Read/Write" (dropdown with "Read/Write"), "Mode" (dropdown with "Realtime"), "Unit" (text input), "Description" (text area), and "Group" (dropdown with "default"). At the bottom right are "Cancel" and "OK" buttons.

**Add Variable** X

\* Variable Name: test5

\* Register Type: DB

\* DB Number: 6

\* Register Address: 18

\* Data Type: FLOAT

Decimal Places: 2

\* Read/Write: Read/Write

\* Mode: Realtime

Unit:

Description:

\* Group: default

Cancel OK

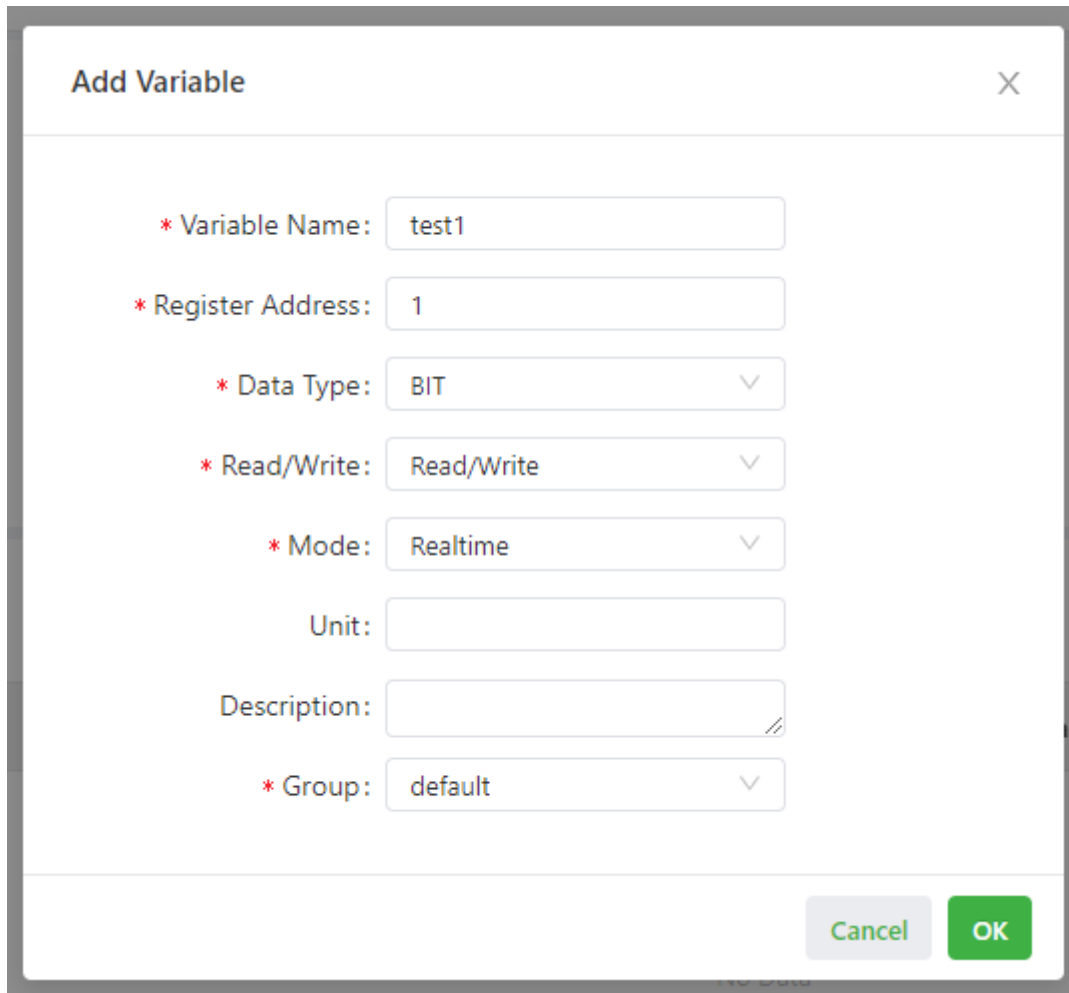
- Add the Modbus variable

On the **Device List** page, click **Add** variable, and configure the variable parameters in the pop-up box for adding variables.

- **Variable Name:** The variable name. (The variable name must be unique in the same device.)
- **Register Address:** The variable register address.
- **Data Type:** The variable data type, including:
  - \* **BOOL:** True or False.
  - \* **BIT:** 0 or 1.
  - \* **WORD:** A 16-bit unsigned character.

- \* INT: A 16-bit signed character.
- \* DWORD: A 32-bit unsigned character.
- \* DINT: A 32-bit signed character.
- \* FLOAT: A 32-bit floating point.
- \* STRING: An 8-bit string.
- **Decimal Places:** The number of decimal places of the variable when the data type is FLOAT. The maximum value is 6.
- **Size:** The string length when the data type is STRING.
- **Register Bit:** The bit offset of the variable when the address is 30001~40000, 310001~365535, 40001~50000, and 410001~465535 and the data type is BOOL or BIT. Any integer from 0 to 15 is supported.
- **Read/write:**
  - \* **Read:** Read only.
  - \* **Write:** Write only.
  - \* **Read/Write:** Read and write.
- **Mode:**
  - \* **Realtime:** Collect the variable data at the collection interval of the group it belongs and report the data at the report interval.
  - \* **Onchange:** Collect the variable data only when the data changes and report the data at the report interval.
- **Unit:** The variable unit.
- **Description:** The variable description.
- **Group:** The collection group to which the variable belongs.

The following figure is an example of adding a coil variable with the address 00001:

A screenshot of a software dialog box titled "Add Variable" with a close button (X) in the top right corner. The dialog contains several input fields, each preceded by a red asterisk indicating it is required. The fields are: "Variable Name" with the text "test1"; "Register Address" with the value "1"; "Data Type" with a dropdown menu showing "BIT"; "Read/Write" with a dropdown menu showing "Read/Write"; "Mode" with a dropdown menu showing "Realtime"; "Unit" with an empty text box; "Description" with an empty text box and a small icon in the bottom right corner; and "Group" with a dropdown menu showing "default". At the bottom right of the dialog are two buttons: "Cancel" and "OK".

**Add Variable** X

\* Variable Name: test1

\* Register Address: 1

\* Data Type: BIT ▾

\* Read/Write: Read/Write ▾

\* Mode: Realtime ▾

Unit:

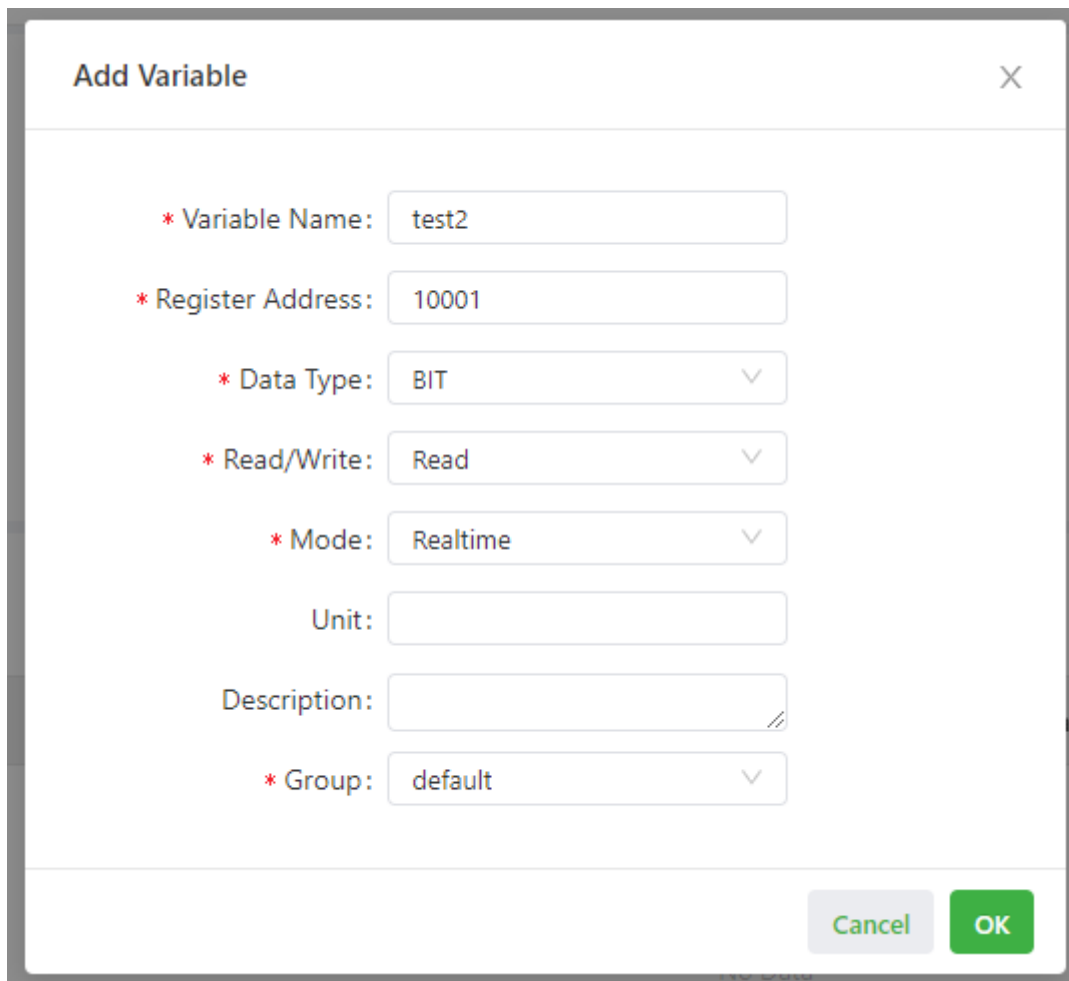
Description:

\* Group: default ▾

Cancel OK

The following figure is an example of adding a switch variable with the address 10001:





The image shows a dialog box titled "Add Variable" with a close button (X) in the top right corner. The dialog contains several input fields and dropdown menus, each preceded by a red asterisk indicating a required field. The fields are: "Variable Name" with the value "test2", "Register Address" with the value "10001", "Data Type" with a dropdown menu showing "BIT", "Read/Write" with a dropdown menu showing "Read", "Mode" with a dropdown menu showing "Realtime", "Unit" with an empty text box, "Description" with an empty text box and a double-slash icon at the end, and "Group" with a dropdown menu showing "default". At the bottom right of the dialog are two buttons: "Cancel" and "OK".

**Add Variable** [X]

\* Variable Name: test2

\* Register Address: 10001

\* Data Type: BIT

\* Read/Write: Read

\* Mode: Realtime

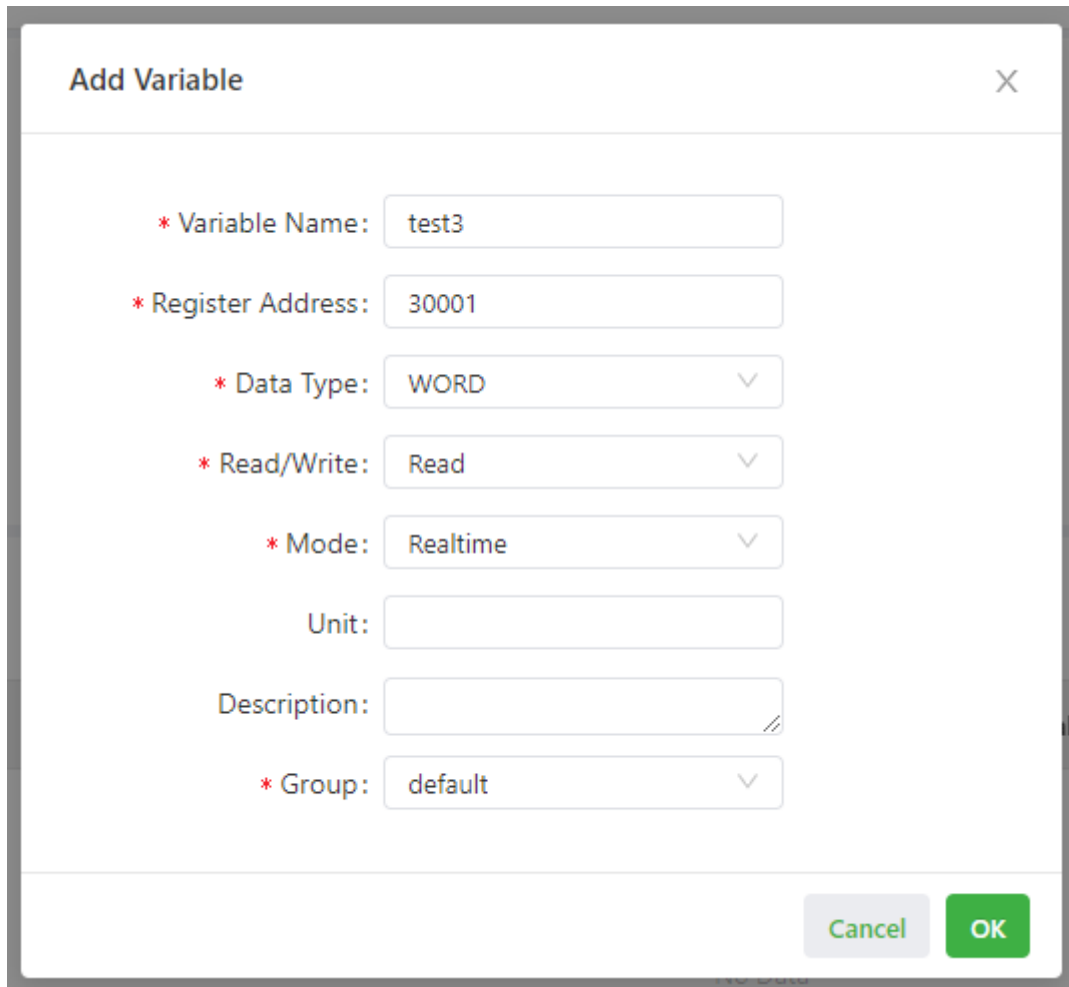
Unit:

Description:

\* Group: default

Cancel OK

The following figure is an example of adding an integer variable with the address 30001:



The image shows a dialog box titled "Add Variable" with a close button (X) in the top right corner. The dialog contains several input fields and dropdown menus, each preceded by a red asterisk indicating a required field. The fields are: "Variable Name" with the value "test3", "Register Address" with the value "30001", "Data Type" with a dropdown menu showing "WORD", "Read/Write" with a dropdown menu showing "Read", "Mode" with a dropdown menu showing "Realtime", "Unit" with an empty text box, "Description" with an empty text box and a small icon in the bottom right corner, and "Group" with a dropdown menu showing "default". At the bottom right of the dialog are two buttons: "Cancel" (light gray) and "OK" (green).

Field	Value
* Variable Name	test3
* Register Address	30001
* Data Type	WORD
* Read/Write	Read
* Mode	Realtime
Unit	
Description	
* Group	default

The following figure is an example of adding a floating point variable with the address 40001:

**Add Variable** [X]

\* Variable Name:

\* Register Address:

\* Data Type:

Decimal Places:

\* Read/Write:

\* Mode:

Unit:

Description:

\* Group:

Cancel OK

- Add EtherNET/IP variable (**App version 1.2.5 and above required**)

Click **Add Variable** button on the **Device List** page. To configure the parameters of EtherNET/IP variable in the add variable pop-up box. It is unnecessary to configure the data type. The Device Supervisor will judge the data type by itself. (Currently supported EIP data types include 'BOOL' 'SINT' 'INT' 'DINT' 'REAL' 'STRING' ):

- Variable name: the name of the variable. (variable name cannot be repeated under the same device)
- Label: The label of the variable.
- Decimal: The length of data after the decimal point of a variable when the data type is a floating point number up to 6 bits.
- Read-write access permission:
  - \* Read: Read only, not writable

- \* Write: Write only, not readable
- \* Read/Write: Read and write
- Collection mode:
  - \* Realtime: Collect variables at fixed collection intervals and report data at reporting intervals.
  - \* Onchange: The data is collected and reported according to the reporting interval after the variable value changes.
- Unit: Unit of a variable.
- Description: Variable description.
- Group: The collection group where the variable belongs.

The following is an example of adding a variable with the label name ZB.LEN.16:

The screenshot shows a dialog box titled "Add Variable" with a close button (X) in the top right corner. The dialog contains the following fields and controls:

- \* Variable Name:** A text input field containing "EIP-test".
- \* Symbol:** A text input field containing "ZB.LEN.16".
- Decimal Places:** A text input field containing "2", followed by a help icon (question mark in a circle).
- \* Read/Write:** A dropdown menu with "Read/Write" selected.
- \* Mode:** A dropdown menu with "Realtime" selected.
- Unit:** An empty text input field.
- Description:** An empty text input field with a double-slash icon (//) at the end.
- \* Group:** A dropdown menu with "default" selected.

At the bottom right of the dialog, there are two buttons: "Cancel" (light gray) and "Confirm" (green).

### 2.2.3 Configure an alarm policy

Choose **Edge Computing > Device Supervisor > Alarm > Alarm Strategy**, click **Add**, and configure the alarm policy parameters in the pop-up box. You can configure the alarm policy in two modes: **Use New Variable** and **Use Exist Variable**. The parameters are described as follows:

- Use New Variable
  - **Name:** The alarm name.
  - **Group:** The alarm group.
  - **Mode:** In Use New Variable mode, the alarm variable is not configured in the Device List. You need to configure the variable parameters, which does not add a new variable to the Device List.
  - **Device:** The device of the alarm variable.
  - **Register Type:** The variable register type. Four types are supported: I/Q/M/DB.(ISO on TCP variable)
  - **Register Address:** The address of the alarm variable.
  - **Data Type:** The data type of the alarm variable.
  - **Alarm Condition**
    - \* **Judgment Conditions:** Valid values are =, !=, >, <, and .
    - \* **Logical Condition**
      - **None:** Judge the alarm based on a single judgment condition.
      - **&&:** Judge the alarm based on two single judgment conditions in AND mode.
      - **||:** Judge the alarm based on two single judgment conditions in OR mode.
  - **Description:** The alarm description.

The following figure is an example of adding an alarm variable. When its value is greater than 30 but less than 50, the alarm is triggered; otherwise, the alarm is not triggered or is cleared.

**Add** [X]

\* Name: Warn1

\* Group: warning

☒ Use New Variable ☐ Use Exist Variable

\* Device: Modbus

\* Register Address: 40001

\* Data Type: WORD

\* Alarm Condition: > 30 && < 50

\* Description: Speed over 30!

Cancel OK

- Use Exist Variable
  - **Name:** The alarm name.
  - **Group:** The alarm group.
  - **Mode:** In Use Existing Variable mode, the alarm variable is already configured in the Device List. You can enter the variable name and use it directly.
  - **Device:** The device of the alarm variable.
  - **Variable Name:** The name of referenced variable.
  - **Alarm Condition**
    - \* **Judgment Conditions:** Valid values are =, !=, >, , <, and .
    - \* **Logical Condition**
      - **None:** Judge the alarm based on a single judgment condition.
      - **&&:** Judge the alarm based on two single judgment conditions in AND mode.

- ||: Judge the alarm based on two single judgment conditions in OR mode.
- **Description:** The alarm description.

The following figure is an example of using an existing variable to generate an alarm variable. When its value is greater than 30 but less than 50, the alarm is triggered; otherwise, the alarm is not triggered or is cleared.

The screenshot shows a dialog box titled "Add" with a close button (X) in the top right corner. The dialog contains the following fields and options:




- \* Name:** Warn1
- \* Group:** warning (dropdown menu)
- Use New Variable** (radio button) and **Use Exist Variable** (radio button, selected)
- \* Device:** Modbus (dropdown menu)
- \* Variable Name:** test4
- \* Alarm Condition:**
  - Condition 1: > 30
  - Condition 2: && (dropdown menu)
  - Condition 3: < 50
- \* Description:** Speed over 30!




At the bottom right, there are two buttons: "Cancel" (grey) and "OK" (green).




## 2.2.4 Configure a group

To configure different collection intervals for a variable or an alarm or report the variable data by MQTT topic, choose **Edge Computing > Device Supervisor > Group**, and add a group on the page.

Overview / Edge Computing / Device Supervisor / Group

Operation :   

<input type="checkbox"/>	Name	Type	Polling Interval(S)	Uploading Interval(S)	Operation 
<input type="checkbox"/>	warning	Alarm	10	--	
<input type="checkbox"/>	default	Collect	10	10	

Total 2 items  **1**  50 / page 

- Add a collection group

The following figure is an example of adding a collection group named **group2**. Data of variables in this group is collected once every 5 seconds.

Add Group

\* Name:

group2

\* Type:

☒ Collect ☐ Alarm

\* Polling Interval:

5

S(1-3600)

\* Uploading Interval:

5

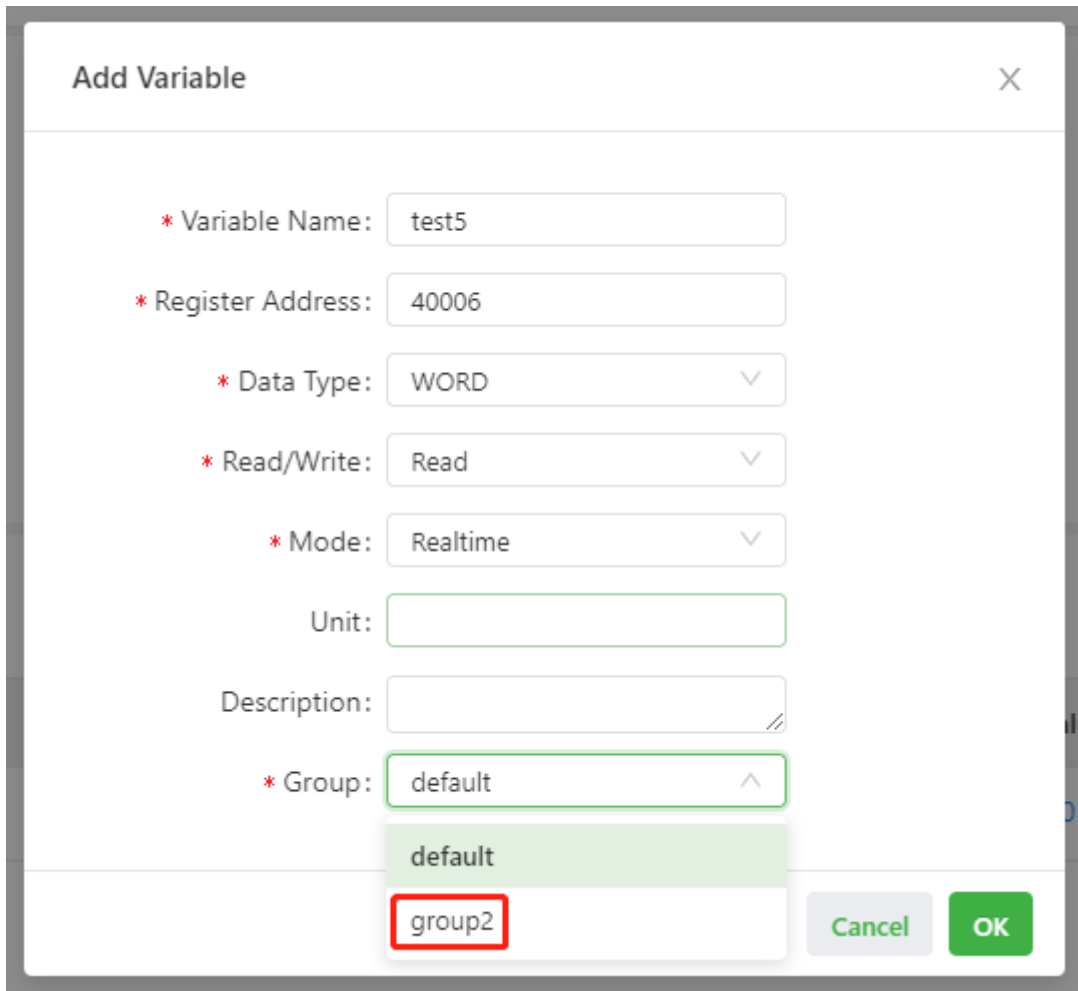
S(1-3600)

Cancel

OK

After a collection group is added, you can associate the variable you added with the group or select the target variable from the variable list and add it to the specified group. Data of variables in the group is collected at the group's collection interval.





**Add Variable** [X]

\* Variable Name:

\* Register Address:

\* Data Type:

\* Read/Write:

\* Mode:

Unit:

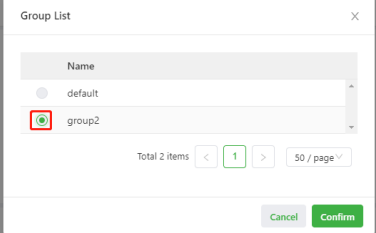
Description:

\* Group:

default

group2

Cancel OK



Group List

Name

default

group2

Total 2 items < 1 > 50 / page

Cancel Confirm

Overview / Edge Computing / Device Supervisor / Device List

Device List

Modbus

ModbusTCP

IP: 10.5.16.77

Variable Table(Modbus)

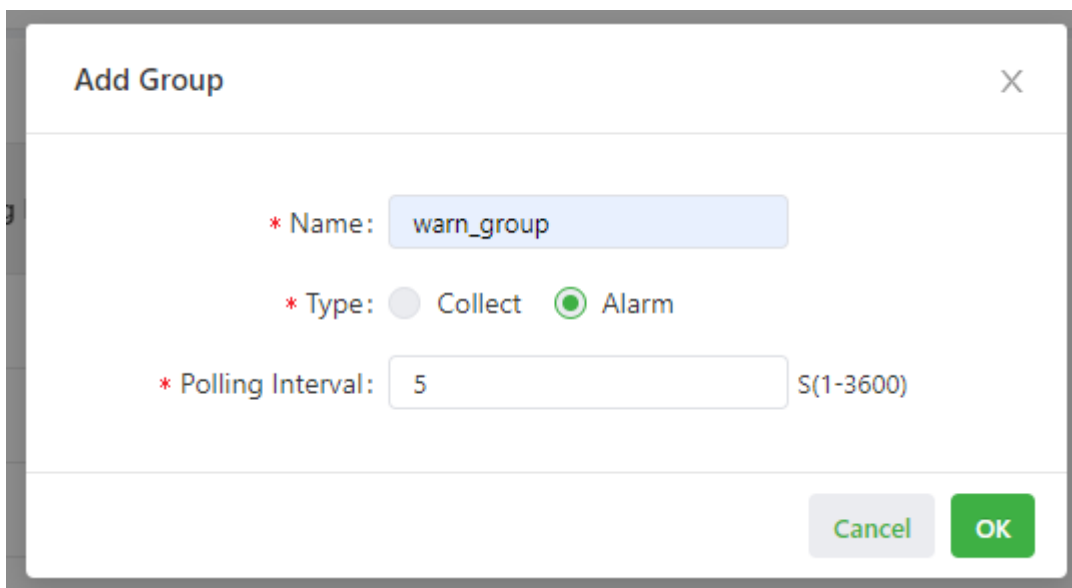
Name	Group	Data Type	Address	Value	Description	Time	Operation
test4	default	FLOAT	40001	0.0		2020-06-02 14:50:58	

Added to Group Delete

Total 1 items < 1 > 50 / page

- Add an alarm group

The following figure is an example of adding an alarm group named warn\_group. The alarm group checks whether the alarm variables in the group are in alarm state once every 5 seconds.

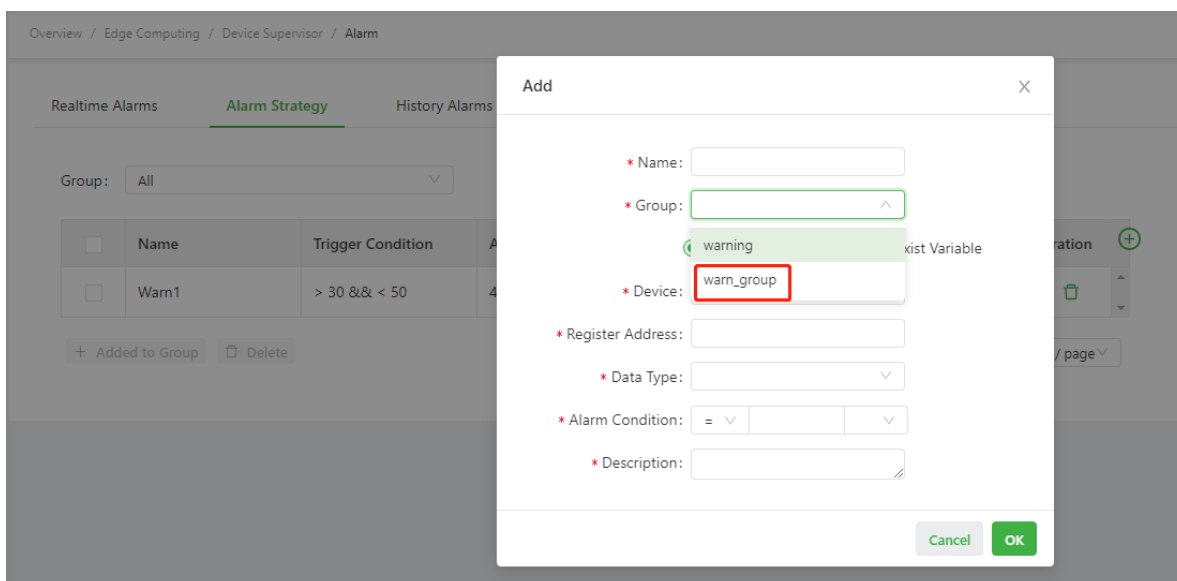


The 'Add Group' dialog box is shown. It has a title bar with 'Add Group' and a close button (X). The form contains the following fields:

- \* Name: warn\_group
- \* Type: ☐ Collect ☒ Alarm
- \* Polling Interval: 5 S(1-3600)

At the bottom right, there are 'Cancel' and 'OK' buttons.

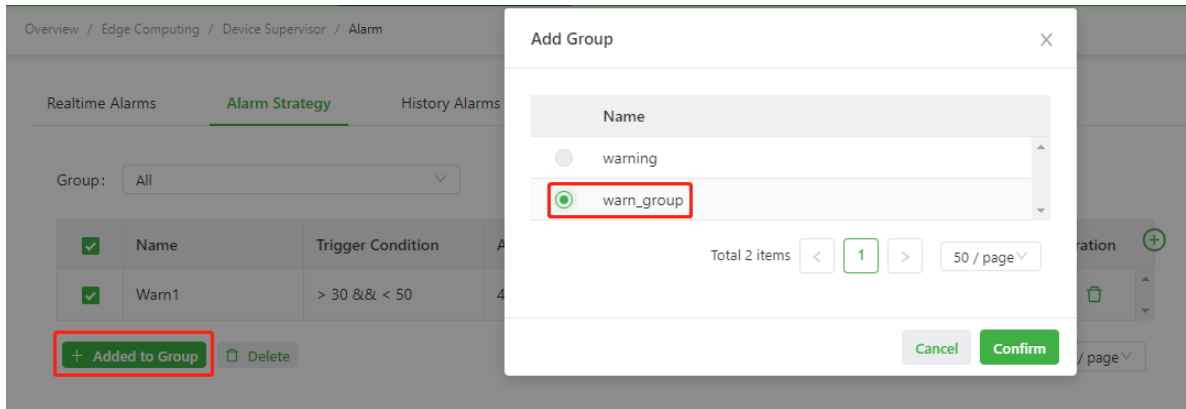
After an alarm group is added, you can associate the alarm policy you added with the group or select the target alarm policy from the alarm list and add it to the specified group. The system checks the variable alarm state at the group's collection interval according to the alarm policies in the group.



The 'Add' dialog box is shown over the 'Alarm Strategy' section. The background shows a table with columns 'Name' and 'Trigger Condition'. The 'Add' dialog box has the following fields:

- \* Name:
- \* Group: warning
- \* Device: warn\_group
- \* Register Address:
- \* Data Type:
- \* Alarm Condition: =
- \* Description:

At the bottom right, there are 'Cancel' and 'OK' buttons.



### 1.1.4 3. Report and monitor the PLC data

- 3.1 Monitor the PLC data locally
- 3.2 Monitor the PLC data on cloud

#### 3.1 Monitor the PLC data locally

- 3.1.1 Monitor data collection locally
- 3.1.2 Monitor the alarm locally

##### 3.1.1 Monitor data collection locally

After configuring data collection, choose **Edge Computing > Device Supervisor > Device List**, and view the data collection status. Click the device card in Device List to switch to the desired PLC data.

**Device List**

View the data of the S7-1200 device

Operation:

Variable Table(S7-1200)

Name	Group	Data Type	Address	Value	Description	Time	Operation
test2	default	FLOAT	DB6.14	528.56		2020-06-15 21:39:37	
T1	default	BOOL	I0.0	false		2020-06-15 21:39:37	
T2	default	BOOL	I0.1	true		2020-06-15 21:39:37	

Blue means that data is collected and gray means that no data is collected.

Blue means writable and gray means non-writable.

The variable value. It is empty if no data is collected.

Collection time of the latest data

Operation:

Total 3 items < 1 > 50 / page

Click the button in the value column to write values.

Overview / Edge Computing / Device Supervisor / Device List

**Device List** Operation:

☐ S7-1200  
ISO-on-TCP  
IP: 10.5.16.73

☐ Modbus  
ModbusTCP  
IP: 10.5.16.77

Total 2 items < 1 >

**Variable Table(S7-1200)** Operation:

<input type="checkbox"/>	Name	Group	Data Type	Address	Value	Description	Time	Operation
<input type="checkbox"/>	test1	default	BOOL	I0.0	false		2020-06-02 15:59:51	
<input type="checkbox"/>	test2	default	FLOAT	DB6.18	12.3		2020-06-02 15:59:51	

+ Added to Group ☐ Delete

Total 2 items < 1 > 50 / page

Overview / Edge Computing / Device Supervisor / Device List

**Device List** Operation:

☐ S7-1200  
ISO-on-TCP  
IP: 10.5.16.73

☐ Modbus  
ModbusTCP  
IP: 10.5.16.77

Total 2 items < 1 >

**Variable Table(S7-1200)** Operation:

<input type="checkbox"/>	Name	Group	Data Type	Address	Value	Description	Time	Operation
<input type="checkbox"/>	test1	default	BOOL	I0.0	false		2020-06-02 16:00:41	
<input type="checkbox"/>	test2	default	FLOAT	DB6.18	13.3		2020-06-02 16:00:41	

+ Added to Group ☐ Delete

Total 2 items < 1 > 50 / page

After the value is modified, the page is as follows:

Overview / Edge Computing / Device Supervisor / Device List

**Modify successful**

**Device List** Operation:

☐ S7-1200  
ISO-on-TCP  
IP: 10.5.16.73

☐ Modbus  
ModbusTCP  
IP: 10.5.16.77

Total 2 items < 1 >

**Variable Table(S7-1200)** Operation:

<input type="checkbox"/>	Name	Group	Data Type	Address	Value	Description	Time	Operation
<input type="checkbox"/>	test1	default	BOOL	I0.0	false		2020-06-02 16:01:21	
<input type="checkbox"/>	test2	default	FLOAT	DB6.18	12.3		2020-06-02 16:01:21	

+ Added to Group ☐ Delete

Total 2 items < 1 > 50 / page

### 3.1.2 Monitor the alarm locally

After configuring the alarm policy, choose **Edge Computing > Device Supervisor > Alarm**, and view the variable alarm status.

- Realtime Alarms: View alarm messages that are not cleared.

Overview / Edge Computing / Device Supervisor / Alarm

Realtime Alarms Alarm Strategy History Alarms

Name	Status	Description	value	Time	Operation
Warn1	Triggered	Speed over 30!	31	2020-06-02 16:05:52	

Total 1 items < 1 > 50 / page

- History Alarms: Screen any alarm messages you want to view.

Overview / Edge Computing / Device Supervisor / Alarm

Realtime Alarms Alarm Strategy History Alarms

Name  Time: 2020-05-03 16:09 ~ 2020-06-02 16:09

Operation:

<input type="checkbox"/>	Name	Status	Description	value	Time	Operation
<input type="checkbox"/>	Warn1	Triggered	Speed over 30!	33.3	2020-06-02 16:09:42	
<input type="checkbox"/>	Warn1	Restored	Speed over 30!	24.5	2020-06-02 16:06:52	
<input type="checkbox"/>	Warn1	Triggered	Speed over 30!	31	2020-06-02 16:05:52	

Total 3 items < 1 > 50 / page

## 3.2 Monitor the PLC data on cloud

- 3.2.1 Configure ThingsBoard
- 3.2.2 Configure a cloud service to report and receive data

### 3.2.1 Configure ThingsBoard

For the usage method of ThingsBoard, see [Get Started with ThingsBoard](#) or refer to *ThingsBoard Reference Flowchart* for test.

### 3.2.2 Configure a cloud service to report and receive data

Choose **Edge Computing > Device Supervisor > Cloud Service**. Select **Enable Cloud Service**, configure the MQTT connection parameters, and then click **Submit**.

- **Type:** The connection of Thingsboard is based on **Standard MQTT**. For more information about how to use **AWS IoT**, please refer to [AWS IoT User Manual](#); For more information about how to use **Azure IoT**, please refer to [Azure IoT User Manual](#).
- **Server Address:** The ThingsBoard demo address is `demo.thingsboard.io`.
- **Client ID:** Any unique ID.
- **Username:** The access token of the ThingsBoard device. For more information about how to obtain the access token, see *Transmit the PLC data to ThingsBoard*.
- **Password:** A password consisting of 6-32 bits.
- Use default values for other configuration items.

After the configuration is completed, the page is as follows:

Overview / Edge Computing / Device Supervisor / Cloud

## Status

Cloud Status: Disconnect

Connection time:

### Enable Cloud Service:



\* Type:

MQTT

\* Server Address:

demo.thingsboard.io

\* Client ID:

datatest

Enable Authority:



\* Username:

24

Password:

.....



### Advanced Settings ▾

\* Port:

1883

\* Keep Alive:

60

s(1-3600)

\* TLS Encryption:



Disable



Enable

\* Clean Session:



NO



YES

\* MQTT Version:



MQTTv31



MQTTv311

Then, click **Messages Management** to configure the publish and subscribe topics. For more information about how to configure the publish and subscribe topics, see *Messages Management (custom MQTT publish/subscribe)*. The following is a configuration example:

- Publish messages:
  - Topic: v1/devices/me/telemetry
  - Qos(MQTT): 1

- Group Type: Collect
- Group: The name of the group whose data needs to be uploaded to ThingsBoard. **default** is used in this manual.
- Main Function: The name of the entry function. **upload\_test** is used in this manual.
- Script:

```
from common.Logger import logger #Import log printing module logger.

def upload_test(data, wizard_api): #Define the main function upload_test.
    logger.info(data) #Print the collected data in logs of the INFO level.
    value_dict = {} #Define the report data dictionary value_dict.
    for device, val_dict in data['values'].items(): #Traverse the values
↪dictionary in the data. The dictionary contains the device name and the
↪variables of the device.
        for id, val in val_dict.items(): #Traverse variables and assign values
↪for the value_dict dictionary.
            value_dict[id] = val["raw_data"]
            value_dict["timestamp"] = data["timestamp"]
        logger.info(value_dict) #Print data of the value_dict dictionary in logs of
↪the INFO level.
    return value_dict #Send value_list to the app, which then sequential
↪uploads it to the MQTT server. The value_list is finally in the following
↪format: {'bool': False, 'byte': 7, 'real': 0.0, 'timestamp': 1583990892.
↪5429199}.
```

After the configuration is completed, the page is as follows:



**Edit Publish**

\* Name:

\* Topic:

\* Qos(MQTT):

Group Type: ☒ Collect ☐ Alarm

\* Group:

\* Main Function:  ⓘ Matches the name of the entry function in the script

\* Script:

```

1  from common.Logger import logger #Import log printing module
2
3  def upload_test(data, wizard_api): #Define the main function
4      logger.info(data) #Print the collected data in logs of
5      value_dict = {} #Define the report data dictionary value
6      for device, val_dict in data['values'].items(): #Traverse
7          for id, val in val_dict.items(): #Traverse variable
8              value_dict[id] = val["raw_data"]
9              value_dict["timestamp"] = data["timestamp"]
10     logger.info(value_dict) #Print data of the value_dict
11     return value_dict #Send value_list to the app, which ti

```

Cancel OK

- Subscribe messages:

- Topic: v1/devices/me/rpc/request/+
- Qos(MQTT): 1
- Main Function: The name of the entry function. `ctl_test` is used in this manual.
- Script:

```

from common.Logger import logger #Import log printing module logger.
import json #Import the JSON module.

def ctl_test(topic, payload, wizard_api): #Define the main function ctl_test.
    logger.info(topic) #Print the subscribe topic.
    logger.info(payload) #Print the subscribe data. ThingsBoard devices deliver
    data in the following format: {"method":"setValue","params":true}.

```

(continues on next page)

(continued from previous page)

```
payload = json.loads(payload) #Deserialize subscribe data.
if payload["method"] == "setValue": #Check whether the data is written.
    message = {"bool":payload["params"]} #Define the message for modifying
↪variables, including the variable names and values.
    ack_tail = [topic.replace('request', 'response'), message] #Define the
↪confirmation data, including the response topic and message.
    wizard_api.write_plc_values(message, ack, ack_tail) #Call the write_plc_
↪values method to deliver data from the message dictionary to the specified
↪variable. Call the ack method and deliver ack_tail to the ack method.

def ack(data, ack_tail, wizard_api): #Define the ack method.
    resp_topic = ack_tail[0] #Define the response topic.
    resp_data = ack_tail[1] #Define the response data.
    wizard_api.mqtt_publish(resp_topic, json.dumps(resp_data), 1) #Call the
↪mqtt_publish method to deliver the response data to the MQTT server in {'bool
↪': True} format.
```

After the configuration is completed, the page is as follows:

**EditSubscribe**

\* Name:

\* Topic:

\* Qos(MQTT):

\* Main Function:  ⓘ Matches the name of the entry function in the script

\* Script:

```

1  from common.Logger import logger #导入打印日志模块logger
2  import json #导入json模块
3
4  def ctl_test(topic, payload, wizard_api): #定义主函数ctl_test
5      logger.info(topic) #打印订阅主题
6      logger.info(payload) #打印订阅数据, Thingsboard的下发数据
7      payload = json.loads(payload) #反序列化订阅数据
8      if payload["method"] == "setValue": #检测是否为写入数据
9          message = {"bool":payload["params"]} #定义修改变量值
10         ack_tail = [topic.replace('request', 'response'), ]
11         wizard_api.write_plc_values(message, ack, ack_tail)
12
13
14  def ack(data, ack_tail, wizard_api): #定义ack方法
15      resp_topic = ack_tail[0] #定义响应主题
16      resp_data = ack_tail[1] #定义响应数据

```

Cancel OK

### 1.1.5 Appendix

- *Importing/exporting data collection configuration*
- *Messages Management (custom MQTT publish/subscribe)*
- *Parameter Settings*
- *Gateway other configuration*
- *ThingsBoard reference flowchart*

#### Importing/exporting data collection configuration

Device Supervisor provides four CSV configuration files for data collection configuration (To use the Alarm policy configuration file function, App version should be 1.2.5 and later). You can quickly configure data collection by importing or exporting configuration files. Each configuration file contains the following items:

- **device.csv:** The device configuration file, which contains the following parameters:
  - **Device Name:** The device name.

- **Protocol:** The communication protocol, such as ModbusTCP.
- **Ip/Serial:** Enter the IP address for Ethernet devices, and enter RS485 or RS232 for serial port devices.
- **Port:** The communication port number of the Ethernet device.
- **Rack (Only ISO on TCP devices):** The rack number of the device.
- **Slot (Only ISO on TCP devices):** The slot number of the device.
- **Mode (Only ISO on TCP devices):** The ISO on TCP mode. Valid values are TSAP and Rack/Slot.
- **Slave (Only Modbus devices):** The address of the slave station.
- **Byte Order (Only Modbus devices):** The byte order. Valid values are abcd, badc, cdab, and dcba.

The export method is to export the device list on the **Device List** page.

Device List

Operation:

Variable Table(S7-1200)

Operation:

<input type="checkbox"/>	Name	Group	Data Type	Address	Value	Description	Time	Operation
<input type="checkbox"/>	test1	default	BOOL	I0.0	false		2020-06-02 16:32:01	
<input type="checkbox"/>	test2	default	FLOAT	DB6.14	24.5		2020-06-02 16:32:01	

Total 2 items 1

50 / page

The following is a configuration example:

Device Name	Protocol	Ip	Port	Slave	Byte Order
Modbus test	ModbusTCP	10.5.16.82	502		1 cdab

- **var.csv:** The variable configuration file, which contains the following parameters:
  - **Var Name:** The variable name.
  - **Device:** The device of the variable.
  - **Protocol:** The communication protocol.
  - **Dbnumber (Only ISO on TCP devices):** The DB number.
  - **Register Type (Only ISO on TCP devices):** The register type, such as DB.
  - **Register Addr:** The register address.
  - **Register Bit:** The bit offset.
  - **Data Type:** The data type.

- **Read Write:** The read/write permission. Valid values are Read/Write, Write, and Read.
- **Float Repr:** The number of decimal places. Valid values are 1~6.
- **Mode:** The collection mode. Valid values are **realtime** and **onchange**.
- **Unit:** The unit.
- **Size:** The string length.
- **Desc:** The description.
- **Group:** The group.

The export method is to export the variable list on the **Device List** page.

Overview / Edge Computing / Device Supervisor / Device List

**Device List**

Operation :

Total 2 items **1**

**Variable Table(S7-1200)**

Operation :

<input type="checkbox"/>	Name	Group	Data Type	Address	Value	Description	Time	Operation
<input type="checkbox"/>	test1	default	BOOL	I0.0	false		2020-06-02 16:33:31	
<input type="checkbox"/>	test2	default	FLOAT	D86.14	24.5		2020-06-02 16:33:31	

+ Added to Group Delete

Total 2 items **1** 50 / page




The following is a configuration example:








A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Var Name	Device	Protocol	Dbnumber	Register Type	Register Addr	Register Bit	Data Type	Read Write	Float Repr	Mode	Unit	Size	Desc	Group
SP1	S7-1200	ISO-on-TCP	0	I	0	0	BOOL	read/write		2	realtime		1	default



- **group.csv:** The group configuration file, which contains the following parameters:
  - **Group Name:** The group name.
  - **Polling Interval:** The collection interval.
  - **Upload Interval:** The upload interval. Left it empty when the group type is **alarm**.
  - **Group Type:** The group type. Valid values are **alarm** and **collect**.

The export method is to export groups on the **Group** page.

Overview / Edge Computing / Device Supervisor / Group

Operation :   

<input type="checkbox"/>	Name	Type	Polling Interval(S)	Uploading Interval(S)	Operation 
<input type="checkbox"/>	warning	Alarm	10	--	
<input type="checkbox"/>	default	Collect	10	10	
<input type="checkbox"/>	group2	Collect	5	5	 
<input type="checkbox"/>	warn_group	Alarm	5	--	 

Total 4 items  1  50 / page

The following is a configuration example:

Group Name	Polling Interval	Upload Interval	Group Type
default	10	10	collect

- **warn.csv**: The alarm policy configuration file, which contains the following parameters:
  - **Warn Name**: The alarm name.
  - **Group**: The group.
  - **Quotes**: Whether to reference the variable. 0 means “use new variable” , 1 means “reference existing variable” .
  - **Device**: The device of the alarm variable.
  - **Var Name**: The referenced variable name. Leave blank when the variable is not referenced.
  - **Condition1**: The alarm condition 1. Eq means “equal to” , Neq means “not equal to” , Gt means “greater than” , Gne means “greater than or equal to” , Lne means “less than or equal to” , Lt means “less than” .
  - **Operand1**: The alarm threshold 1.
  - **Combine Method**: The alarm condition connection mode. None means empty, And means &&, Or means ||
  - **Condition2**: The alarm condition 2.
  - **Operand2**: The alarm threshold 2.
  - **Alarm Content**: The alarm description.
  - **Register Addr**: The alarm variable address.


- **Dbnumber:** The DB number of the variable when the alarm variable register type is DB.
- **Data Type:** The alarm variable data type. Leave it blank when configuring EtherNET/IP and OPCUA variables.
- **Symbol:** The alarm variable tag name. Need to fill in when configuring EtherNET/IP variables.
- **Register Type:** The alarm variable register type.
- **Register Bit:** The bit offset of the variable when the data type of the alarm variable is BOOL or BIT.
- **Namespace Index:** The namespace index when the alarm variable is the OPCUA protocol.
- **Identifier:** The identification when the alarm variable is the OPCUA protocol.
- **Identifier Type:** The ID type when the alarm variable is the OPCUA protocol.
- **Float Repr:** The number of decimal places.




The export method is to export alarms on the **Alarm** page.

Overview / Edge Computing / Device Supervisor / Alarm

---

Realtime Alarms **Alarm Strategy** History Alarms

All Operation : 

<input type="checkbox"/>	Name	Trigger Condition	Device	Target Address	Description	Group	Operation 
<input type="checkbox"/>	Warn1	= 1	EIP	ZB.LEN.16	Test	warning	 

+ Add to Group ☐ Delete

Total 1 item < 1 > 50 / page

## Messages Management (custom MQTT publish/subscribe)

You can choose **Edge Computing > Device Supervisor > Cloud Service** and configure MQTT connection parameters. You can configure the MQTT messages, data source, and other parameters for the data to be uploaded, and customize the data upload and processing logic for the MQTT publish and subscribe topics in Python. In this way, you can perform data upload and delivery with multiple types of MQTT servers without secondary development. The following describes how to use **Messages Management**.

- *Configure Publish Messages*
- *Configure Subscribe Messages*
- *Device Supervisor API Description*
- *Device Supervisor API Callback Function Description*

## Configure Publish Messages

To customize a publish message, configure the following items:

- **Name:** The custom publish name.
- **Topic:** The publish topic, which must be consistent with the topic subscribed to by the MQTT server.
- **Qos(MQTT):** The publish QoS, which is recommended to be consistent with that of the MQTT server.
  - 0: The message is sent only once, without retry.
  - 1: The message is sent at least once to ensure that it reaches the MQTT server.
  - 2: Ensure that the message reaches the MQTT server and the MQTT server receives it only once.
- **Group Type:** When publishing variable data, select **Collect**, and then only **Collect Group** is available in Group. When publishing alarm data, select **Alarm**, and then only **Alarm Group** is available in Group.
- **Group:** After a group is selected, data of all variables in this group is uploaded to the MQTT server based on this publish configuration. You can select multiple groups.
- **Main Function:** The name of the main function (entry function), which must be consistent with that in the script.
- **Script:** Use the Python code to customize the packaging and processing logic. Main functions in publish include the following parameters:
  - **Parameter 1:** Device Supervisor sends the collected variable data to this parameter in the following format:
    - \* Variable data format:

```
{
    'timestamp': 1589434519.5458372, #The timestamp when data is generated.
    'group_name': 'default', #The name of the collect group.
    'values': #The variable data dictionary, including the PLC name,
    ↪variable name, and variable value.
    {
        'S7-1200': #The PLC name.
        {
            'Test1': #The variable name.
            {
                'raw_data': False, #The variable value.
                'status': 1 #The collection status. If the value is not 1,
                ↪the collection is abnormal.
            },
            'Test2':
            {
                'raw_data': 2,
                'status': 1
            }
        }
    }
}
```

(continues on next page)



(continued from previous page)

```

    }
  }
}

```

\* Alarm data format:

```

{
  'timestamp': 1589434527.3628697, #The timestamp when an alarm is
↳generated.
  'group_name': 'warning', #The name of the alarm group.
  'values': #The alarm data dictionary, including the alarm information
↳such as alarm name.
  {
    'Warn1': #The alarm name.
    {
      'timestamp': 1589434527, #The timestamp when an alarm is
↳generated.
      'current': 'on', #The alarm status. on: The alarm has been
↳triggered. off: The alarm has been cleared.
      'status': 0, #The alarm status. 0: The alarm has been
↳triggered. 1: The alarm has been cleared.
      'value': 33, #The value of the alarm variable when the alarm
↳is triggered.
      'alarm_content': 'The speed has exceeded 30.', #The alarm
↳description.
      'level': 1 #The reserved field.
    }
  }
}

```

- **Parameter 2:** It is the API provided by Device Supervisor. For more information about it, see *Device Supervisor API Description*.

The following are examples of common custom publish methods. Do not use the `mqtt_publish` or `save_data` method together with the `return` command:

- Publish example 1: Upload the variable data in `return` mode

In this example, the variable data is uploaded in `return` mode, the processed variable data is sent to Device Supervisor through the `return` command. Device Supervisor automatically sequential uploads the variable data by collection time to the MQTT server according to the topic and QoS configured in the publish. If the variable data failed to send, it caches the variable data, waits until the MQTT

connection is normal, and then sequential uploads the variable data by collection time to the MQTT server. The following is an example of publish and code configuration:

**Edit Publish**

\* Name:

\* Topic:

\* Qos(MQTT):

Group Type: ☒ Collect ☐ Alarm

\* Group:

\* Main Function:  ⓘ Matches the name of the entry function in the script

\* Script:

```

6 """
7
8 def vars_upload_test(data_collect, wizard_api): #Define the
9     value_list = [] #Define the data list.
10     for device, val_dict in data_collect['values'].items():
11         value_dict = { #Customize the data dictionary.
12             "Device": device,
13             "timestamp": data_collect["timestamp"],
14             "Data": {}
15         }
16         for id, val in val_dict.items(): #Traverse variable
17             value_dict["Data"][id] = val["raw_data"]
18         value_list.append(value_dict) #Add data in value_d
19     logging.info(value_list) #Print data in value_list in
20     return value_list #Send value_list to the app, which t
  
```

Cancel OK

```

import logging
"""
Logs are generally printed in the gateway in the following ways:
1.import logging: Use logging.info(XXX) to print logs. Logs printed in this way are
↳not controlled by the log level parameter on the Parameter Settings page.
2.from common.Logger import logger: Use logger.info(XXX) to print logs. Logs
↳printed in this way are controlled by the log level parameter on the Parameter
↳Settings page.
"""

def vars_upload_test(data_collect, wizard_api): #Define the main function for
↳publish.
  
```

(continues on next page)

(continued from previous page)

```

value_list = [] #Define the data list.
for device, val_dict in data_collect['values'].items(): #Traverse the values
↪dictionary. The dictionary contains the device name and the variables of the
↪device.
    value_dict = { #Customize the data dictionary.
        "Device": device,
        "timestamp": data_collect["timestamp"],
        "Data": {}
    }

    for id, val in val_dict.items(): #Traverse variables and assign values for
↪the Data dictionary.
        value_dict["Data"][id] = val["raw_data"]
    value_list.append(value_dict) #Add data in value_dict to value_list in
↪sequence.

    logging.info(value_list) #Print data in value_list in app logs in the following
↪format: [{'Device': 'S7-1200', 'timestamp': 1589538347.5604711, 'Data': {'Test1':
↪False, 'Test2': 12}}].

    return value_list #Send value_list to the app, which then sequential uploads it
↪to the MQTT server by collection time. If it fails to be sent, cache the data and
↪sequential upload it to the MQTT server by collection time after the connection
↪recovers.

```

- Publish example 2: Upload the alarm data in **return** mode

In this example, alarm data is uploaded. The following is an example of publish and code configuration:

Edit Publish
X

\* Name: default

\* Topic: a1/xxx/yyy

\* Qos(MQTT): 1

Group Type: ☐ Collect ☒ Alarm

\* Group: warning X

\* Main Function: alarms\_upload\_test ⓘ Matches the name of the entry function in the script

\* Script:

```

1  import logging
2  """
3  Logs are generally printed in the gateway in the following
4  1.import logging: Use logging.info(XXX) to print logs. Logs
5  2.from common.Logger import logger: Use logger.info(XXX) to
6  """
7
8  def alarms_upload_test(data_collect, wizard_api): #Define
9      alarm_list = [] #Define the alarm list.
10     for alarm_name, alarm_info in data_collect['values'].i
11         alarm_dict = { #Customize the data dictionary.
12             "Alarm_name": alarm_name,
13             "timestamp": data_collect["timestamp"],
14             "Alarm_status": alarm_info['current'],
15             "Alarm_value": alarm_info['value'],
16             "Alarm content": alarm_info['alarm c

```

Cancel OK

```

import logging
"""
Logs are generally printed in the gateway in the following ways:
1.import logging: Use logging.info(XXX) to print logs. Logs printed in this way are
↳not controlled by the log level parameter on the Parameter Settings page.
2.from common.Logger import logger: Use logger.info(XXX) to print logs. Logs
↳printed in this way are controlled by the log level parameter on the Parameter
↳Settings page.
"""

def alarms_upload_test(data_collect, wizard_api): #Define the main function for
↳publish.
    alarm_list = [] #Define the alarm list.
    for alarm_name, alarm_info in data_collect['values'].items(): #Traverse the
↳values dictionary. The dictionary contains the information such as alarm name and
↳alarm generation time.

```

(continues on next page)

(continued from previous page)

```

alarm_dict = { #Customize the data dictionary.
    "Alarm_name": alarm_name,
    "timestamp": data_collect["timestamp"],
    "Alarm_status": alarm_info['current'],
    "Alarm_value": alarm_info['value'],
    "Alarm_content": alarm_info['alarm_content']
}

alarm_list.append(alarm_dict) #Add data in alarm_dict to alarm_list in
↪sequence.

logging.info(alarm_list) #Print alarm_list in app logs.

return alarm_list #Send alarm_list to the app, which then sequential uploads it
↪to the MQTT server by collection time. If it fails to be sent, cache the data and
↪sequential upload it to the MQTT server by time after the connection recovers.

```

- Publish example 3: Use `mqtt_publish` to upload the variable data and use `save_data` to save the variable data that failed to upload.

In this example, the variable data is uploaded to the MQTT server through `mqtt_publish`. If the variable data failed to upload due to MQTT connection failure, the topic, QoS, and the variable data are saved to the database through `save_data`. The saved variable data will be uploaded by saving order to the MQTT server based on the topic and QoS in the data after the MQTT connection is resumed to normal. The following is an example of publish and code configuration:

Edit Publish
X

\* Name: default

\* Topic: v1/xxx/yyyy

\* Qos(MQTT): 1

Group Type: ☒ Collect ☐ Alarm

\* Group: default X

\* Main Function: vars\_cache\_test ⓘ Matches the name of the entry function in the script

\* Script:

```

1  from common.Logger import logger
2  import json
3  from datetime import datetime
4  """
5  Logs are generally printed in the gateway in the following
6  1.import logging: Use logging.info(XXX) to print logs. Logs
7  2.from common.Logger import logger: Use logger.info(XXX) to
8  """
9
10 def vars_cache_test(data_collect, wizard_api): #Define the
11     value_list = [] #Define the data list.
12     utc_time = datetime.utcnow().timestamp()
13     for device, val_dict in data_collect['values'].items():
14         value_dict = { #Customize the data dictionary.
15             "DeviceSN": device,
16             "Time": utc_time.strftime('%Y-%m-%dT%

```

Cancel OK

```

from common.Logger import logger
import json
from datetime import datetime
"""
Logs are generally printed in the gateway in the following ways:
1.import logging: Use logging.info(XXX) to print logs. Logs printed in this way are
↳not controlled by the log level parameter on the Parameter Settings page.
2.from common.Logger import logger: Use logger.info(XXX) to print logs. Logs
↳printed in this way are controlled by the log level parameter on the Parameter
↳Settings page.
"""

def vars_cache_test(data_collect, wizard_api): #Define the main function for
↳publish.

```

(continues on next page)

(continued from previous page)

```

value_list = [] #Define the data list.
utc_time = datetime.utcnow().timestamp() #Convert the
↳Linux timestamp to the UTC time.
    for device, val_dict in data_collect['values'].items(): #Traverse the values
↳dictionary. The dictionary contains the device name and the variables of the
↳device.
        value_dict = { #Customize the data dictionary.
            "DeviceSN": device,
            "Time": utc_time.strftime('%Y-%m-%dT%H:%M:%S.%fZ'),
            "Data": {}
        }
        for id, val in val_dict.items(): #Traverse variables and assign values for
↳the Data dictionary.
            value_dict["Data"][id] = val["raw_data"]
        value_list.append(value_dict) #Add data in value_dict to value_list in
↳sequence.
        if not wizard_api.mqtt_publish("v1/xxx/yyy", json.dumps(value_list), 1): #Call
↳the mqtt_publish method of the wizard_api module to send value_list to the MQTT
↳server based on the topic "v1/xxx/yyy" and QoS level 1, and then check whether it
↳is successfully sent.
            value_list = {"topic": "v1/xxx/yyy", "qos": 1, "payload": value_list}
            wizard_api.save_data(value_list) #If it fails to be sent, cache the data
↳and upload it to the MQTT server by time after the connection recovers.
            logger.info("Save data:%s" %value_list)
        logger.info(value_list) #Print value_list in app logs.

```

- Publish example 4: Use `mqtt_publish` to upload the variable data and use `save_data` to save the variable data that failed to upload.

In this example, the variable data is uploaded to the MQTT server through `mqtt_publish`. If the variable data failed to upload due to MQTT connection failure, the variable data and group name are saved through `save_data`. The saved variable data will be uploaded by saving order to the MQTT server based on the topic and QoS associated with the group in the cloud service after the MQTT connection is resumed to normal. Do not use the `mqtt_publish` or `save_data` method together with the `return` command. The following is an example of publish and code configuration:

Edit Publish

×

\* Name: default

\* Topic: v1/xxx/yyyy

\* Qos(MQTT): 1

Group Type: ☒ Collect ☐ Alarm

\* Group: default ×

\* Main Function: vars\_cache\_test ⓘ Matches the name of the entry function in the script

\* Script:

```

1  from common.Logger import logger
2  import json
3  from datetime import datetime
4  """
5  Logs are generally printed in the gateway in the following
6  1.import logging: Use logging.info(XXX) to print logs. Logs
7  2.from common.Logger import logger: Use logger.info(XXX) to
8  """
9
10 def vars_cache_test(data_collect, wizard_api): #Define the
11     value_list = [] #Define the data list.
12     utc_time = datetime.utcnow()
13     for device, val_dict in data_collect['values'].items():
14         value_dict = { #Customize the data dictionary.
15             "DeviceSN": device,
16             "Time": utc_time.strftime('%Y-%m-%dT%

```

Cancel

OK

```

from common.Logger import logger
import json
from datetime import datetime
"""
Logs are generally printed in the gateway in the following ways:
1.import logging: Use logging.info(XXX) to print logs. Logs printed in this way are
↳not controlled by the log level parameter on the Parameter Settings page.
2.from common.Logger import logger: Use logger.info(XXX) to print logs. Logs
↳printed in this way are controlled by the log level parameter on the Parameter
↳Settings page.
"""

def vars_cache_test(data_collect, wizard_api): #Define the main function for
↳publish.

```

(continues on next page)

56

Chapter 1. InGateway Documentation Site Navigation



(continued from previous page)

```

value_list = [] #Define the data list.
utc_time = datetime.utcnow().timestamp() #Convert the
↳Linux timestamp to the UTC time.
    for device, val_dict in data_collect['values'].items(): #Traverse the values
↳dictionary. The dictionary contains the device name and the variables of the
↳device.
        value_dict = { #Customize the data dictionary.
            "DeviceSN": device,
            "Time": utc_time.strftime('%Y-%m-%dT%H:%M:%S.%fZ'),
            "Data": {}
        }
        for id, val in val_dict.items(): #Traverse variables and assign values for
↳the Data dictionary.
            value_dict["Data"][id] = val["raw_data"]
        value_list.append(value_dict) #Add data in value_dict to value_list in
↳sequence.
        if not wizard_api.mqtt_publish("v1/xxx/yyy", json.dumps(value_list), 1): #Call
↳the mqtt_publish method of the wizard_api module to send value_list to the MQTT
↳server based on the topic "v1/xxx/yyy" and QoS level 1, and then check whether it
↳is successfully sent.
            wizard_api.save_data(value_list, 'default') #If it fails to be sent, cache
↳the data and upload it to the MQTT server by time after the connection recovers.
            logger.info("Save data:%s" %value_list)
            logger.info(value_list) #Print value_list in app logs.

```

- Publish example 5: Use `get_tag_config` to get the configuration of devices, variables, and alarms.

In this example, when the app is restarted each time, the configuration of devices, variables, and alarms are gotten through `get_tag_config` respectively and then sent to the MQTT server(This example is only useful for getting Rackslot pattern point tables for ISO on TCP and Modbus). The following is an example of publish and code configuration:

Edit Publish

\* Name:

\* Topic:

\* Qos(MQTT):

Group Type:

☒ Collect
 ☐ Alarm

\* Group:

\* Main Function:

Matches the name of the entry function in the script

\* Script:

```

1  from common.Logger import logger
2  import json
3  """
4  Logs are generally printed in the gateway in the following
5  1.import logging: Use logging.info(XXX) to print logs. Log:
6  2.from common.Logger import logger: Use logger.info(XXX) to
7  """
8
9  IS_UPLOAD_CONFIG = True #Define a variable to determine w
10
11 def upload_tagconfig(recv, wizard_api): #Define the main
12     global IS_UPLOAD_CONFIG #Declare that the variable is
13     if IS_UPLOAD_CONFIG: #Determine whether to acquire and
14         wizard_api.get_tag_config(tagconfig) #Call the re
15         IS_UPLOAD_CONFIG = False #Do not upload the confi
16

```

Cancel

OK

```

from common.Logger import logger
import json
"""

```

Logs are generally printed in the gateway in the following ways:

- 1.import logging: Use logging.info(XXX) to print logs. Logs printed in this way are ↵  
↵not controlled by the log level parameter on the Parameter Settings page.
- 2.from common.Logger import logger: Use logger.info(XXX) to print logs. Logs ↵  
↵printed in this way are controlled by the log level parameter on the Parameter ↵  
↵Settings page.

```

"""

```

```

IS_UPLOAD_CONFIG = True #Define a variable to determine whether to acquire and ↵
↵upload the configuration.

```

(continues on next page)

(continued from previous page)

```

def upload_tagconfig(recv, wizard_api): #Define the main function for publish.
    global IS_UPLOAD_CONFIG #Declare that the variable is a global variable.
    if IS_UPLOAD_CONFIG: #Determine whether to acquire and upload the
↪ configuration.
        wizard_api.get_tag_config(tagconfig) #Call the recall_data method of the
↪ wizard_api module to define the callback function name of the method as tagconfig.
        IS_UPLOAD_CONFIG = False #Do not upload the configuration again after it
↪ is acquired and uploaded.

def tagconfig(config, tail, wizard_api): #Define the callback function tagconfig
↪ used to acquire the configuration.
    logger.info(config) #Print the configuration information, including the device,
↪ group, variable, and alarm configuration.
    deviceConfiguration_list = [] #Define the list of device configuration.
    for device in config['devices']: #Traverse the device configuration.
        deviceInfo = {} #Define the device information dictionary.
        if device['protocol'] == "ModbusTCP": #Determine whether the device
↪ communication protocol is ModbusTCP.
            deviceInfo["Device"] = device['device_name']
            deviceInfo["PLCProtocol"] = device['protocol']
            deviceInfo["IP Address"] = device['ip']
            deviceInfo["Port"] = device['port']
            deviceInfo["SlaveAddress"] = device['slave']
            deviceInfo["Endian"] = device['byte_order']
            deviceConfiguration_list.append(deviceInfo) #Add deviceInfo to
↪ deviceConfiguration_list in sequence.
            elif device['protocol'] == "ModbusRTU": #Determine whether the device
↪ communication protocol is ModbusRTU.
                deviceInfo["Device"] = device['device_name']
                deviceInfo["PLCProtocol"] = device['protocol']
                deviceInfo["Port"] = device['serial']
                deviceInfo["Baudrate"] = device['baudrate']
                deviceInfo["DataBits"] = device['bytesize']
                deviceInfo["Parity"] = device['parity']
                deviceInfo["StopBits"] = device['stopbits']
                deviceInfo["SlaveAddress"] = device['slave']
                deviceInfo["Endian"] = device['byte_order']
                deviceConfiguration_list.append(deviceInfo)
            elif device['protocol'] == "ISO-on-TCP": #Determine whether the device
↪ communication protocol is ISO-on-TCP.

```

(continues on next page)

(continued from previous page)

```

        deviceInfo["Device"] = device['device_name']
        deviceInfo["PLCProtocol"] = device['protocol']
        deviceInfo["IP Address"] = device['ip']
        deviceInfo["Port"] = device['port']
        deviceInfo["Rack"] = device['rack']
        deviceInfo["Slot"] = device['slot']
        deviceConfiguration_list.append(deviceInfo)

    logger.info(deviceConfiguration_list)
    wizard_api.mqtt_publish("Config/DeviceInfo", json.dumps(deviceConfiguration_
↪list), 1) #Call the mqtt_publish method of the wizard_api module to send_
↪deviceConfiguration_list to the MQTT server based on the topic "Config/DeviceInfo
↪" and QoS level 1.

    tagConfiguration_list = [] #Define the list of variable configuration.
    device_group_info_dict = {} #Define the group information dictionary of a_
↪device.
    group_info_dict = {} #Define the group information dictionary.
    for groupinfo in config['groups']: #Traverse the groups in the configuration.
        group_info_dict[groupinfo["group_name"]] = groupinfo #Map the group names_
↪and group information.
    for device in config['devices']: #Traverse the devices in the configuration.
        group_list= [] #Define the group list of a device.
        for var in config['vars']: #Traverse the variables in the configuration.
            if device['device_name'] == var['device'] and var['group'] not in group_
↪list: #Determine whether the variable is in the device. If the group of the_
↪variable is not in group_list, add the group to group_list.
                group_list.append(var['group'])
        device_group_info_dict[device['device_name']] = group_list #Map the devices_
↪and the group lists of the devices.
    for device, group_list in device_group_info_dict.items(): #Traverse the group_
↪information dictionary of a device.
        if group_list == []: #If the group list of the device is empty, no variable_
↪is defined for the device. Then, skip this device.
            continue
        tagConfiguration = {} #Define the variable configuration dictionary.
        tagConfiguration["Device"] = device #Add device information.
        tagConfiguration["Collections"] = []
        for group in group_list: #Traverse the groups of a device and add group_
↪information.

```

(continues on next page)

(continued from previous page)

```

group_info = {}
group_info["CollectionName"] = group
group_info["SampleRate"] = group_info_dict[group]["polling_interval"]
group_info["PublishInterval"] = group_info_dict[group]["upload_interval"]
↪"]

group_info["TagData"] = []
tagConfiguration["Collections"].append(group_info)
for var in config['vars']: #Traverse the variables in the configuration.
    if var['device'] != device or var['group'] != group: #If the
↪variable does not belong to the device and group, skip this variable.
        continue
    index_number = tagConfiguration["Collections"].index(group_info)
↪#Obtain the index of the group in tagConfiguration["Collections"].
    data_info = {} #Define the variable information dictionary.
    data_info["Tag"] = var["var_name"]
    data_info["Address"] = var["address"]
    data_info["ValueType"] = var["data_type"]
    data_info["AccessLevel"] = var["read_write"]
    data_info["Mode"] = var["mode"]
    data_info["Unit"] = var["unit"]
    data_info["Description"] = var["desc"]
    tagConfiguration["Collections"][index_number]["TagData"].
↪append(data_info) #Add variable information to TagData of the specified group.
    tagConfiguration_list.append(tagConfiguration) #Add the tagConfiguration to
↪tagConfiguration_list in sequence.
    logger.info(tagConfiguration_list) #Print the variable Configuration.
    for tagConfiguration in tagConfiguration_list: #Traverse the variable
↪Configuration of each device and upload them to the MQTT server.
        wizard_api.mqtt_publish("Config/TagConfiguration", json.
↪dumps(tagConfiguration), 1) #Call the mqtt_publish method of the wizard_api
↪module to send tagConfiguration_list to the MQTT server based on the topic
↪"Config/TagConfiguration" and QoS level 1.

alarmConfiguration_list = [] #Define the alarm configuration.
for alarm in config['warning']: #Traverse the alarms in the configuration.
    alarmInfo = {} #Define the alarm information dictionary.
    alarmInfo['Warn_name'] = alarm['warn_name']
    alarmInfo['Group'] = alarm['group']
    alarmInfo['Alarm_content'] = alarm['alarm_content']

```

(continues on next page)

(continued from previous page)

```

alarmInfo['Condition1'] = alarm['condition1']
alarmInfo['Operand1'] = alarm['operand1']
alarmInfo['Combine_method'] = alarm['combine_method']
alarmInfo['Condition2'] = alarm['condition2']
alarmInfo['Operand2'] = alarm['operand2']
alarmInfo['Device'] = alarm['device']
alarmInfo['Var_name'] = alarm['var_name']
alarmInfo['Address'] = alarm['address']
alarmConfiguration_list.append(alarmInfo) #Add the alarm information to
↪alarmConfiguration_list in sequence.
logger.info(alarmConfiguration_list) #Print the alarm configuration.
wizard_api.mqtt_publish("Config/AlarmInfo", json.dumps(alarmConfiguration_list),
↪ 1) #Call the mqtt_publish method of the wizard_api module to send
↪alarmConfiguration_list to the MQTT server based on the topic "Config/AlarmInfo"
↪and QoS level 1.







```

- Publish example 6: Use `get_global_parameter` to get custom parameter set in Parameter Settings.

In this example, custom parameter set in **Parameter Settings** are gotten through `device_id`, and the MQTT topic is configured with the wildcard `${device_id}`. The following is an example of publish and code configuration:

[Overview](#) / [Edge Computing](#) / [Device Supervisor](#) / [Global Parameters](#)

## Global Parameters

Parameters	Parameters Value	Operation 
catch_recording	100000	
device_id	1	 
log_level	INFO	
warning_recording	2000	

Edit Publish
X

\* Name: default

\* Topic: global/\${device\_id}/parameter

\* Qos(MQTT): 1

Group Type: ☒ Collect ☐ Alarm

\* Group: default X

\* Main Function: vars\_upload\_test ⓘ Matches the name of the entry function in the script

\* Script:

```

1  import logging
2  """
3  Logs are generally printed in the gateway in the following
4  1.import logging: Use logging.info(XXX) to print logs. Logs
5  2.from common.Logger import logger: Use logger.info(XXX) to
6  """
7
8  def vars_upload_test(data_collect, wizard_api): #Define the
9      global_parameter = wizard_api.get_global_parameter() #
10     logging.info(global_parameter) #Print the global variab
11     value_list = [] #Define the data list.
12     for device, val_dict in data_collect['values'].items():
13         value_dict = { #Customize the data dictionary.
14             "Device": device,
15             "DeviceID": global_parameter["device
16             "timestamp": data_collect["timestamp"]

```

Cancel OK

```

import logging
"""
Logs are generally printed in the gateway in the following ways:
1.import logging: Use logging.info(XXX) to print logs. Logs printed in this way are
↳not controlled by the log level parameter on the Parameter Settings page.
2.from common.Logger import logger: Use logger.info(XXX) to print logs. Logs
↳printed in this way are controlled by the log level parameter on the Parameter
↳Settings page.
"""

def vars_upload_test(data_collect, wizard_api): #Define the main function for
↳publish.
    global_parameter = wizard_api.get_global_parameter() #Define the Custom
↳Parameter variables.

```

(continues on next page)

(continued from previous page)

```

logging.info(global_parameter) #Print the Custom Parameter variables.
value_list = [] #Define the data list.
for device, val_dict in data_collect['values'].items(): #Traverse the values
↪ dictionary. The dictionary contains the device name and the variables of the
↪ device.
    value_dict = { #Customize the data dictionary.
        "Device": device,
        "DeviceID": global_parameter["device_id"], #Acquire the
↪ device ID defined in Custom Parameter.
        "timestamp": data_collect["timestamp"],
        "Data": {}
    }

    for id, val in val_dict.items(): #Traverse variables and assign values for
↪ the Data dictionary.
        value_dict["Data"][id] = val["raw_data"]
    value_list.append(value_dict) #Add data in value_dict to value_list in
↪ sequence.

    logging.info(value_list) #Print data in value_list in app logs in the following
↪ format: [{'Device': 'S7-1200', 'DeviceID': '1', 'timestamp': 1589538347.5604711,
↪ 'Data': {'Test1': False, 'Test2': 12}}].

    return value_list #Send value_list to the app, which then sequential uploads it
↪ to the MQTT server by collection time. If it fails to be sent, cache the data and
↪ sequential upload it to the MQTT server by time after the connection recovers.

```

## Configure Subscribe Messages

Custom subscribe message contains the following items:

- **Name:** The custom subscribe name.
- **Topic:** The subscribe topic, which must be consistent with the topic published by the MQTT server.
- **Qos(MQTT):** The subscribe QoS, which is recommended to be consistent with that of the MQTT server.
- **Main Function:** The name of the main function (entry function), which must be consistent with that in the script.
- **Script:** Use the Python code to custom the packaging and processing logic. Main functions in subscribe include the following parameters:
  - **Parameter 1:** It is the received topic. The data type is `string`.
  - **Parameter 2:** It is the received data. The data type is `string`.



- **Parameter 3:** It is the API provided by Device Supervisor. For more information about it, see *Device Supervisor API Description*.

The following are four common custom subscribe methods:

- Subscribe example 1: Deliver the variable name and value and write the PLC data but do not return the write result.

In this example, a command is delivered by the MQTT server to modify the variable value. The following is an example of publish and code configuration:

**Edit Subscribe**

\* Name:

\* Topic:

\* Qos(MQTT):

\* Main Function:  ⓘ Matches the name of the entry function in the script

\* Script:

```

1  import logging
2  import json
3
4  def ctl_test(topic, payload, wizard_api): #Define the main
5      logging.info(topic) #Print the subscribe topic. Assume
6      logging.info(payload) #Print the subscribe data. Assume
7      payload = json.loads(payload) #Deserialize subscribe d
8      if payload["method"] == "setValue": #Check whether the
9          message = {payload["TagName"]:payload["TagValue"]}
10         wizard_api.write_plc_values(message) #Call the writ

```

Cancel OK

```

import logging
import json

def ctl_test(topic, payload, wizard_api): #Define the main function for subscribe.
    logging.info(topic) #Print the subscribe topic. Assume that the topic is "write/
    ↪ plc".
    logging.info(payload) #Print the subscribe data. Assume that the payload data
    ↪ is {"method":"setValue", "TagName":"SP1", "TagValue":12.3}.
    payload = json.loads(payload) #Deserialize subscribe data.

```

(continues on next page)

(continued from previous page)

```

if payload["method"] == "setValue": #Check whether the data is written.
    message = {payload["TagName"]:payload["TagValue"]} #Define the message to
    ↳ be delivered, including the variable name and value to be delivered.
    wizard_api.write_plc_values(message) #Call the write_plc_values method of
    ↳ the wizard_api module to deliver data from the message dictionary to the
    ↳ specified variable.

```

- Subscribe example 2: Deliver the device name, and variable name and value and write the PLC data but do not return the write result.

In this example, a command is delivered by the MQTT server to modify the variable value. The following is an example of publish and code configuration:

**Edit Subscribe**

\* Name:

\* Topic:

\* Qos(MQTT):

\* Main Function:  ⓘ Matches the name of the entry function in the script

\* Script:

```

1  import logging
2  import json
3
4  def ctl_test(topic, payload, wizard_api): #Define the main
5      logging.info(topic) #Print the subscribe topic.
6      logging.info(payload) #Print the subscribe data.
7      #Assume that the payload data is {"method":"setValue",
8      payload = json.loads(payload) #Deserialize subscribe d
9      data_dict = {payload["TagName"]:payload["TagValue"]} #
10     var_device = payload["Device"] #Define the name of the
11     if payload["method"] == "setValue": #Check whether the
12         message = {var_device:data_dict} #Define the messa
13         wizard_api.write_plc_values(message) #Call the writ

```

Cancel OK

```

import logging
import json

def ctl_test(topic, payload, wizard_api): #Define the main function for subscribe.

```

(continues on next page)

(continued from previous page)

```

logging.info(topic) #Print the subscribe topic.
logging.info(payload) #Print the subscribe data.
#Assume that the payload data is {"method":"setValue","Device":"Modbus_test",
↪ "TagName":"SP1", "TagValue":12.3}.
payload = json.loads(payload) #Deserialize subscribe data.
data_dict = {payload["TagName"]:payload["TagValue"]} #Define the data
↪ dictionary to be delivered, including the variable name and value to be delivered.
var_device = payload["Device"] #Define the name of the device.
if payload["method"] == "setValue": #Check whether the data is written.
    message = {var_device:data_dict} #Define the message to be delivered,
↪ including the device name and data dictionary to be delivered.
    wizard_api.write_plc_values(message) #Call the write_plc_values method of
↪ the wizard_api module to deliver data from the message dictionary to the
↪ specified variable.

```

- Subscribe example 3: Write the variable data and return the write result.

In this example, a command is delivered by the MQTT server to modify the variable value and return the modification result. The following is an example of publish and code configuration:

Edit Subscribe

X

\* Name:

\* Topic:

\* Qos(MQTT):

\* Main Function:

ⓘ Matches the name of the entry function in the script

\* Script:

```

9      data_dict = {payload["TagName"]:payload["TagValue"]} #Define the data
10     var_device = payload["Device"] #Define the name of the device
11     if payload["method"] == "setValue": #Check whether the data is written.
12         message = {var_device:data_dict} #Define the message to be delivered,
13         ack_tail = [topic.replace('request', 'response'), message] #Define the
14         logging.info(message)
15         wizard_api.write_plc_values(message, ack, ack_tail)
16
17     def ack(send_result, ack_tail, wizard_api): #Define the callback function
18         topic = ack_tail[0] #Define the response topic.
19         if isinstance(send_result,tuple): #Check whether the data is written.
20             resp_data = {"Status":"timeout", "Data":ack_tail[1]}
21         else:
22             resp_data = {"Status":send_result[0]["result"], "Data":send_result[0]}
23         wizard_api.mqtt_publish(topic, json.dumps(resp_data), 0)

```

Cancel

OK

```

import logging
import json

def ctl_test(topic, payload, wizard_api): #Define the main function for subscribe.
    logging.info(topic) #Print the subscribe topic. Assume that the topic is
    ↳ "request/v1".
    logging.info(payload) #Print the subscribe data.
    #Assume that the payload data is {"method":"setValue","Device":"Modbus_test",
    ↳ "TagName":"SP1", "TagValue":12.3}.
    payload = json.loads(payload) #Deserialize subscribe data.
    data_dict = {payload["TagName"]:payload["TagValue"]} #Define the data
    ↳ dictionary to be delivered, including the variable name and value to be delivered.
    var_device = payload["Device"] #Define the name of the device.
    if payload["method"] == "setValue": #Check whether the data is written.
        message = {var_device:data_dict} #Define the message to be delivered,
    ↳ including the device name and data dictionary to be delivered.
        ack_tail = [topic.replace('request', 'response'), message] #Define the
    ↳ confirmation data, including the response topic and message.

```

(continues on next page)

68

Chapter 1. InGateway Documentation Site Navigation

(continued from previous page)

```

        logging.info(message)
        wizard_api.write_plc_values(message, ack, ack_tail, timeout = 0.5) #Call
↪ the write_plc_values method of the wizard_api module to deliver data from the
↪ message dictionary to the specified variable. Define the callback function of
↪ this method as ack and deliver ack_tail to the ack function.

def ack(send_result, ack_tail, wizard_api): #Define the callback function ack.
    topic = ack_tail[0] #Define the response topic.
    if isinstance(send_result,tuple): #Check whether the data type of send-result
↪ is tuple. If so, the delivery times out.
        resp_data = {"Status":"timeout", "Data":ack_tail[1]} #Define the response
↪ for delivery timeout.
    else:
        resp_data = {"Status":send_result[0]["result"], "Data":ack_tail[1]} #Define
↪ the response if the delivery does not time out.
        wizard_api.mqtt_publish(topic, json.dumps(resp_data), 0) #Call the mqtt_publish
↪ method of the wizard_api module to deliver the response data to the MQTT server.

```

- Subscribe example 4: Recall the data immediately.

In this example, when the specified command is delivered by the MQTT server, the system immediately reads values of all variables and sends them to the MQTT server. The following is an example of publish and code configuration:

Edit Subscribe
X

\* Name: Recall data

\* Topic: recall/v1

\* Qos(MQTT): 1

\* Main Function: recall\_test
Matches the name of the entry function in the script

\* Script:

```

1  from common.Logger import logger
2  import json
3
4  def recall_test(topic, payload, wizard_api): #Define the main function for recall_test
5      logger.info(topic) #Print the subscribe topic. Assume that the topic is "recall/v1".
6      payload = json.loads(payload) #Deserialize subscribe data. Assume that the payload data is {"command": "Upload immediately"}.
7      logger.info(payload) #Print the subscribe data. Assume that the payload data is {"command": "Upload immediately"}.
8      if payload["command"] == "Upload immediately": #Check whether to call back the data.
9          wizard_api.recall_data(recall) #Call the recall_data method of the wizard_api module to define the callback function name of the method as recall.
10
11 def recall(data_collect, tail, wizard_api): #Define the callback function recall.
12     logger.info(data_collect) #Print the read data.
13     value_list = [] #Define the data list.
14     for device, val_dict in data_collect["values"].items():
15         value_dict = { #Customize the data dictionary.
16             "DeviceSN": device,

```

Cancel OK

```

from common.Logger import logger
import json

def recall_test(topic, payload, wizard_api): #Define the main function for recall_test
    logger.info(topic) #Print the subscribe topic. Assume that the topic is "recall/v1".
    payload = json.loads(payload) #Deserialize subscribe data. Assume that the payload data is {"command": "Upload immediately"}.
    logger.info(payload) #Print the subscribe data. Assume that the payload data is {"command": "Upload immediately"}.
    if payload["command"] == "Upload immediately": #Check whether to call back the data.
        wizard_api.recall_data(recall) #Call the recall_data method of the wizard_api module to define the callback function name of the method as recall.

def recall(data_collect, tail, wizard_api): #Define the callback function recall.
    logger.info(data_collect) #Print the read data.

```

(continues on next page)

(continued from previous page)

```

value_list = [] #Define the data list.
for device, val_dict in data_collect["values"].items(): #Traverse the values_
↪ dictionary. The dictionary contains the device name and the variables of the_
↪ device.
    value_dict = { #Customize the data dictionary.
                  "DeviceSN": device,
                  "timestamp": data_collect["timestamp"],
                  "Data": []
                }
    for id, val in val_dict.items(): #Traverse variables and assign values for_
↪ the Data list.
        var_dict = {} #Define the variable dictionary.
        var_dict[id] = val["raw_data"]
        value_dict["Data"].append(var_dict) #Add variable dictionaries to value_
↪ dict in sequence.
    value_list.append(value_dict) #Add data dictionaries to value_list in_
↪ sequence.
    logger.info(value_list) #Print value_list.
    wizard_api.mqtt_publish("v1/xxx/yyy", json.dumps(value_list), 1) #Call the mqtt_
↪ publish method of the wizard_api module to send value_list to the MQTT server_
↪ based on the topic "v1/xxx/yyy" and QoS level 1.

```

## Device Supervisor API Description

The API provided by Device Supervisor supports the following methods:

- **mqtt\_publish**: The MQTT message publish method. It is used to send the specified data to the MQTT server based on the topic and return the sending result (True or False). For its usage example, see *Publish example 3*. This method contains the following parameters:
  - **Parameter 1**: The MQTT topic. The data type is **string**. This topic is used to send the data to the MQTT server.
  - **Parameter 2**: The data to be sent.
  - **Parameter 3**: The QoS level. Valid values are 0, 1, and 2.
- **save\_data**: The method for saving data to the database. The saved data will be uploaded by saving order to the MQTT server when the MQTT connection is resumed to normal. For its usage example, see *Publish example 3* and *Publish example 4*. This method contains the following parameters:
  - **Parameter 1**: The data to be saved. If only **Parameter 1** is provided when **save\_data** is called, the data type of **Parameter 1** is **dict**, the saved data must have key-value pairs with **topic**, **qos**,

and `payload` as the keys, and the saved data is sent to the MQTT server based on its `topic`, `qos`, and `payload` after the MQTT connection is resumed to normal.

- **Parameter 2** (`group` is optional): The group name of the data to be saved. The data type is `string`. When `save_data` is called, if **Parameter 2** is provided, **Parameter 1** is sent to the MQTT server based on the topic and QoS associated with this group in the cloud service.
- **write\_plc\_values**: Deliver the data to the specified variable method and return the modification result. For its usage example, see *Subscribe example 1*, *Subscribe example 2* and *Subscribe example 3*. This method contains the following parameters:

- **Parameter 1**: Deliver the data. Two forms are supported:
  - \* **Form 1**: Input a `dict` which uses the variable name and value as the key-value pair. When using this method to modify the variable value, keep the variable name unique in the **Device list**. The following is a data format example:

```
{
    "SP1": 12.3,  #The key-value pair of the variable name and value.
    "SP2": 12.4
}
```

- \* **Form 2**: Input a `dict` that contains the device name and variable name and value. The following is a data format example:

```
{
    "S7-1200":  #The name of the device.
    {
        "SP1": 12.3,  #The key-value pair of the variable name and value.
        "SP2": 12.4
    }
}
```

- **Parameter 2** (`callback` is optional): The name of the callback function that returns the modification result. For more information about the callback function, see *Description of the write\_plc\_values callback function*.
- **Parameter 3** (`tail` is optional): When **Parameter 2** is available, you can assign the data that needs to be sent to the callback function returning the modification result to **Parameter 3**.
- **Parameter 4** (`timeout` is optional): The write timeout time. The data type is `Integer` or `float`. The default value is 60s.
- **get\_tag\_config**: The method to get the configurations, including the PLC, variable, group, and alarm configurations. For its usage example, see *Publish example 5*. This method contains the following parameters:



- **Parameter 1:** The name of the callback function that gets the configuration. For more information about the callback function, see *Description of the get\_tag\_config callback function*.
- **Parameter 2 (tail is optional):** It can be used to assign the data that needs to be sent to the callback function getting the configuration to **Parameter 2**.
- **Parameter 3 (timeout is optional):** The timeout time for getting the configuration. The data type is **Integer**. The default value is 60s.
- **recall\_data:** The method that is used to read values of all variables immediately. For its usage example, see *Subscribe example 4*. This method contains the following parameters:
  - **Parameter 1:** The name of the callback function that immediately reads values of all variables. For more information about the callback function, see *Description of the recall\_data callback function*.
  - **Parameter 2 (tail is optional):** It can be used to assign the data that needs to be sent to the callback function immediately reading values of all variables to **Parameter 2**.
  - **Parameter 3 (timeout is optional):** The timeout time for immediately reading values of all variables. The data type is **Integer**. The default value is 60s.
- **get\_global\_parameter:** The method to get the custom parameter. For its usage example, see *Publish example 6*. This method returns a dictionary for custom parameter. The data format is as follows:

```
{
    'gateway_sn': 'GT902XXXXXXXXXX', #The gateway serial number, which is a system
    ↪parameter.
    'log_level': 'INFO', #The log level, which is a system parameter.
    'catch_recording': 100000, #The maximum number of MQTT messages of variables
    ↪that can be cached.
    'warning_recording': 2000, #The maximum number of MQTT messages of alarms that
    ↪can be cached.
    'device_id': '1' #The custom parameter.
}
```

## Device Supervisor API Callback Function Description

- **write\_plc\_values** Callback function description**write\_plc\_values**The callback function contains the following parameters. For its usage example, see *Subscribe example 3*:
  - **Parameter 1:** The write result of the **write\_plc\_values** method.
  - \* When write times out, the returned value is

```
("error", -110, "timeout")
```

- \* When write succeeds, the format of the returned value is:

```
[
{
    'value': 12, #The written value.
    'device': 'S7-1200', #The written device.
    'var_name': 'Test2', #The written variable name.
    'result': 'OK', #The written result. OK: writing succeeded; Failed:↵
↵writing failed.
    'error': '' #The writing error. When the writing result is OK, this↵
↵parameter is empty.
}]
```

- \* When write failed, the format of the returned value is:

```
[
{
    'value': 12.3,
    'device': 'Modbus_test',
    'var_name': 'SP1',
    'result': 'Failed',
    'error': "Device 'Modbus_test' not found."
}]
```

- Parameter 2: The Parameter 3 configured in the `write_plc_values` method. If Parameter 3 is not configured in `write_plc_values`, this parameter is `None`.
- Parameter 3: It is the API provided by Device Supervisor. For more information about it, see *Device Supervisor API Description*.
- `get_tag_config` Callback function description`get_tag_config`The callback function contains the following parameters. For its usage example, see *Publish example 5*:
  - Parameter 1: The configuration returned by the `get_tag_config` method. When getting configuration times out, the returned value is `("error", -110, "timeout")`; otherwise, the data format is as follows (Take Rack/slot mode of the ISO-on-TCP protocol as an example):

```
{
    'devices': [ #The device configuration.
    {
        'protocol': 'ISO-on-TCP', #The device protocol.
        'device_name': 'S7-1200', #The name of the device.
        'ip': '10.5.16.73', #The IP address.
        'port': 102, #Port
```

(continues on next page)

(continued from previous page)

```

        'rack': 0, #The rack number.
        'slot': 0, #The slot number.
        'id': '6358f50294dc11ea8d890018050ff046' #The device ID.
    }],
    'groups': [ #The group configuration.
    {
        'group_name': 'warning', #The group name.
        'polling_interval': 10, #The collection interval.
        'upload_interval': '', #The reporting interval.
        'group_type': 'alarm', #The group type. collect: collect group; alarm:
↪alarm group.
        'id': '84c371902eb911eabab11a4f32d1ee44' #The group ID.
    }],
    'warning': [ #The alarm configuration.
    {
        'warn_name': 'Warn1', #The alarm name.
        'group': 'warning', #The group of the alarm.
        'quotes': 1, #The variable source of the alarm. 0: direct use address;
↪1: reference address.
        'device': 'S7-1200', #The device of the alarm variable.
        'alarm_content': 'The speed has exceeded 30!' #The alarm description.
        'condition1': 'Gt', #The alarm condition 1. Eq: equal to; Neg: not
↪equal to; Gt: grater than; Gne: greater than or equal to; Lne: less than or
↪equal to; Lt: less than.
        'operand1': '30', #The alarm threshold 1.
        'combine_method': 'And', #The connection mode of alarm conditions.
↪None: empty; And: &&; Or: ||
        'condition2': 'Lt', #The alarm condition 2.
        'operand2': '50', #The alarm threshold 2.
        'var_name': 'Test2', #The variable name of the alarm.
        'var_id': '96c93c3094dd11eabd400018050ff046', #The variable ID of the
↪alarm.
        'size': 1, #The length of the string when the data type of the alarm
↪variable is STRING.
        'float_repr': 2, #The number of digits following the decimal point of
↪the variable when the data type of the alarm variable is FLOAT.
        'id': '9165ed78943e11ea8a000018050ff046', #The alarm ID.
        'address': 'DB6.2', #The variable address of the alarm.
        'protocol': 'ISO-on-TCP', #The communication protocol of the alarm
↪device.
    }
    ]
  }
}

```

(continues on next page)

(continued from previous page)

```

        'data_type': 'WORD', #The data type of the alarm variable.
        'register_type': 'DB', #The register type of the alarm variable.
        'register_addr': 2, #The register address of the alarm variable.
        'read_write': 'read/write', #The read/write permission of the alarm.↵
↵read: read-only; write: write-only; read/write: read and write.
        'mode': 'realtime', #The variable collection mode of the alarm.
        'unit': '', #The variable unit of the alarm.
        'desc': '', #The variable description of the alarm.
        'dbnumber': 6, #The DB number of the variable when the register type of↵
↵the alarm variable is DB.
        'register_bit': '' #The offset of the variable when the data type of↵
↵the alarm variable is BOOL or BIT.
    }],
    'vars': [ #The variable configuration.
        {
            'device': 'S7-1200', #The name of the device to which the variable↵
↵belongs.
            'protocol': 'ISO-on-TCP', #The communication protocol of the device to↵
↵which the variable belongs.
            'data_type': 'BOOL', #The data type of the variable.
            'register_type': 'I', #The register type of the variable.
            'var_name': 'Test1', #The variable name.
            'register_addr': 0, #The register address of the variable.
            'read_write': 'read/write', #The read/write permission of the variable.↵
↵read: read-only; write: write-only; read/write: read and write.
            'mode': 'realtime', #The collection mode of the variable.
            'unit': '', #The unit of the variable.
            'desc': '', #The description of the variable.
            'group': 'default', #The group of the variable.
            'register_bit': 0, #The offset of the variable when the data type of↵
↵the variable is BOOL or BIT.
            'size': 1, #The length of the string when the data type of the variable↵
↵is STRING.
            'float_repr': 2, #The length of the data following the decimal point of↵
↵the variable when the data type of the variable is FLOAT.
            'dbnumber': 0, #The DB number of the variable when the register type of↵
↵the variable is DB.
            'id': 'a1d9439a94dc11eaa2830018050ff046', #The ID of the variable.
            'address': 'I0.0' #The address of the variable.
        },
    ],

```

(continues on next page)

(continued from previous page)

```

{
    'device': 'S7-1200',
    'protocol': 'ISO-on-TCP',
    'data_type': 'WORD',
    'register_type': 'DB',
    'var_name': 'Test2',
    'register_addr': 2,
    'read_write': 'read/write',
    'mode': 'realtime',
    'unit': '',
    'desc': '',
    'group': '2222',
    'dbnumber': 6,
    'size': 1,
    'float_repr': 2,
    'register_bit': '',
    'id': '96c93c3094dd11eabd400018050ff046',
    'address': 'DB6.2'
}]
}

```

- **Parameter 2:** The **Parameter 3** configured in the `get_tag_config` method. If **Parameter 3** is not configured in `get_tag_config`, this parameter is `None`.
- **Parameter 3:** It is the API provided by Device Supervisor. For more information about it, see *Device Supervisor API Description*.
- **recall\_data** Callback function description **recall\_data** The callback function contains the following parameters. For its usage example, see *Subscribe example 4*:
  - **Parameter 1:** The variable data returned by the `recall_data` method. When getting variable data times out, the returned value is `("error", -110, "timeout")`; otherwise, the data format is as follows:

```

{
    'timestamp': 1589507333.2521989, #The timestamp when data is generated.
    'values': #The data dictionary, including the PLC name, variable name, and
    ↪variable value.
    {
        'S7-1200': #The PLC name.
        {
            'Test1': #The variable name.

```

(continues on next page)

(continued from previous page)

```
{
    'raw_data': False, #The variable value.
    'status': 1 #The collection status. If the value is not 1, the
↪collection is abnormal.
},
'Test2':
{
    'raw_data': 33,
    'status': 1
}
}
```

- **Parameter 2:** The **Parameter 3** configured in the `recall_data` method. If **Parameter 3** is not configured in `recall_data`, this parameter is `None`.
- **Parameter 3:** It is the API provided by Device Supervisor. For more information about it, see *Device Supervisor API Description*.

## Parameter Settings

You can choose **Edge Computing > Device Supervisor > Parameter Settings** and configure global settings for Device Supervisor.

- Default Parameter







You can set the log level, the number of historical alarm and historical data in the default parameter.

- Custom Parameter

You can add common parameter to the custom parameter as wildcards and use them in the cloud service. The usage is `${Parameter Name}`, as shown in the figure below:

[Overview](#) / [Edge Computing](#) / [Device Supervisor](#) / [Global Parameters](#)

## Global Parameters

Parameters	Parameters Value	Operation 
catch_recording	100000	
device_id	1	 
log_level	INFO	
warning_recording	2000	

- Serial port settingsIn the serial port setting module, you can configure communication parameters for the serial ports RS485 and RS232, as shown in the figure below:

### Serial Port Settings

RS-485 Serial Port

\* Baud Rate:

\* Data Bits:

\* Parity:

\* Stop Bits:

RS-232 Serial Port

\* Baud Rate:

\* Data Bits:

\* Parity:

\* Stop Bits:

## Gateway other configuration

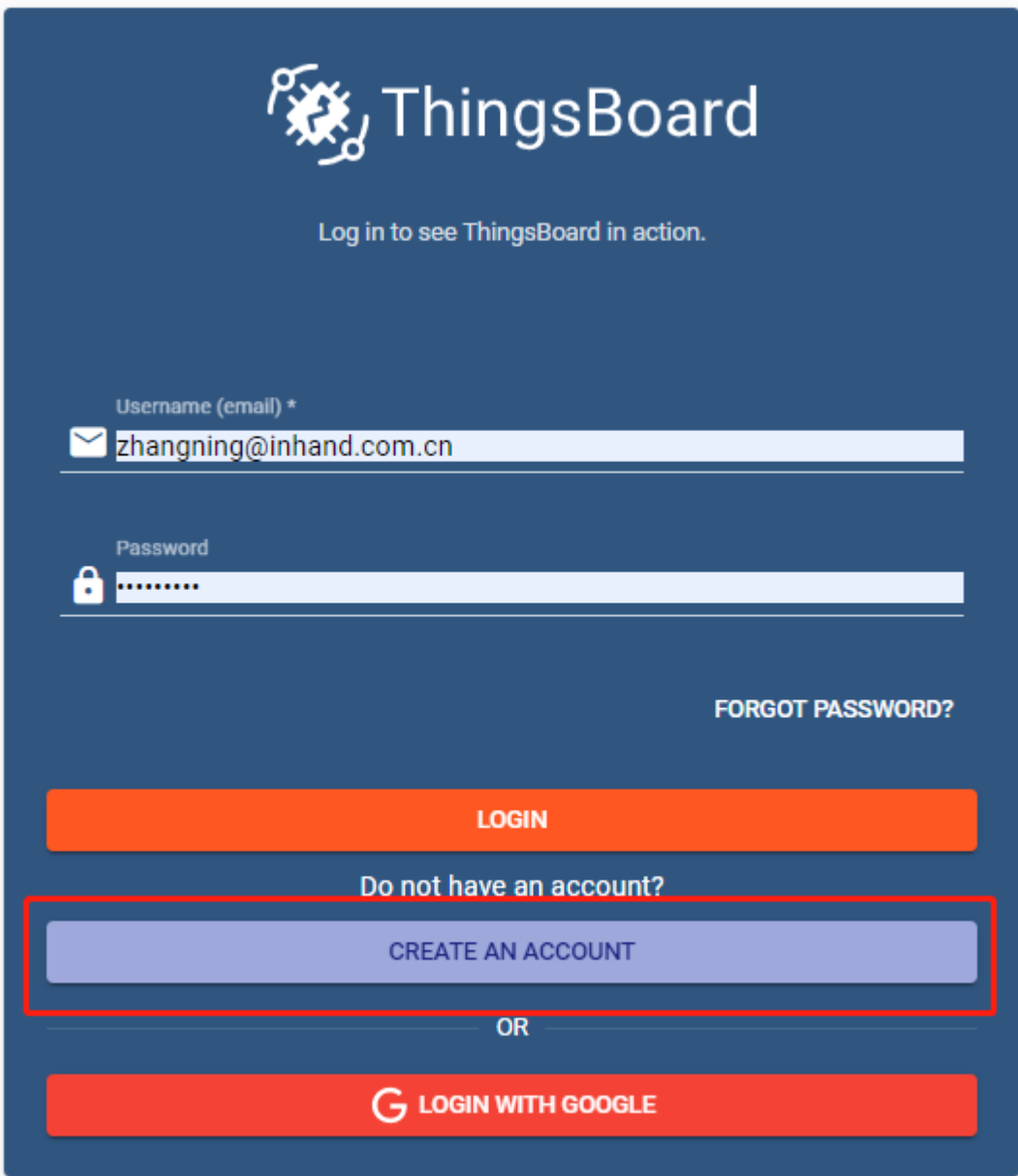
For more information about other common gateway operations, see [Get Started with IG501](#), [Get Started with IG502](#) or [Get Started with IG902](#).

## ThingsBoard reference flowchart

- Add devices and assets*
- Transmit the PLC data to ThingsBoard devices*
- Configure a visual dashboard*

## Add devices and assets

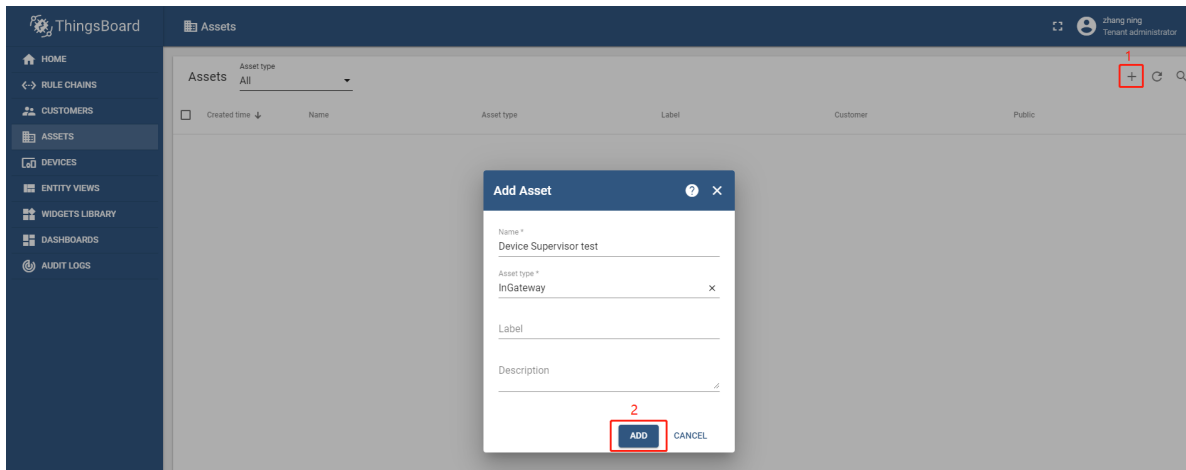
Visit <https://demo.thingsboard.io/login> and enter the login account and password. If you do not have an account yet, register an account first before login. Registration requires access to the network outside China.



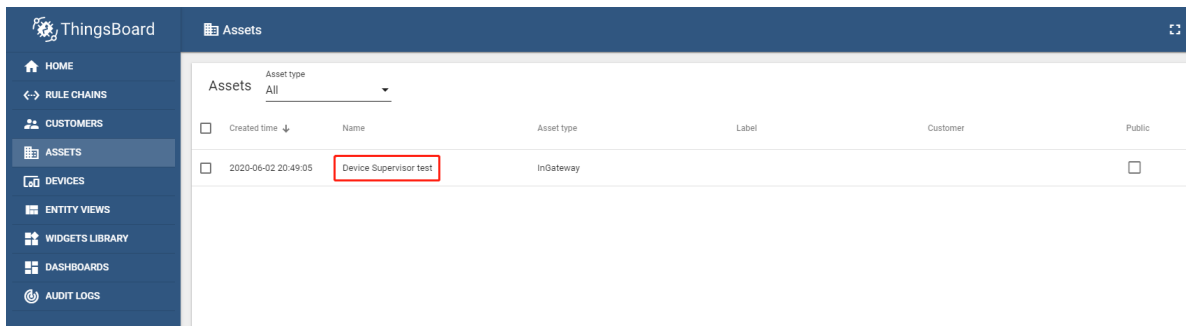
The image shows the ThingsBoard login and registration interface. At the top, the ThingsBoard logo is displayed. Below it, the text "Log in to see ThingsBoard in action." is shown. The login form consists of two input fields: "Username (email) \*" with the value "zhangning@inhand.com.cn" and "Password" with masked characters. To the right of the password field is a link "FORGOT PASSWORD?". Below the login fields is an orange "LOGIN" button. Underneath the login button is the text "Do not have an account?". Below this text is a light blue "CREATE AN ACCOUNT" button, which is highlighted with a red rectangular border. Below the "CREATE AN ACCOUNT" button is the text "OR". At the bottom is a red "G LOGIN WITH GOOGLE" button.

- Add an asset

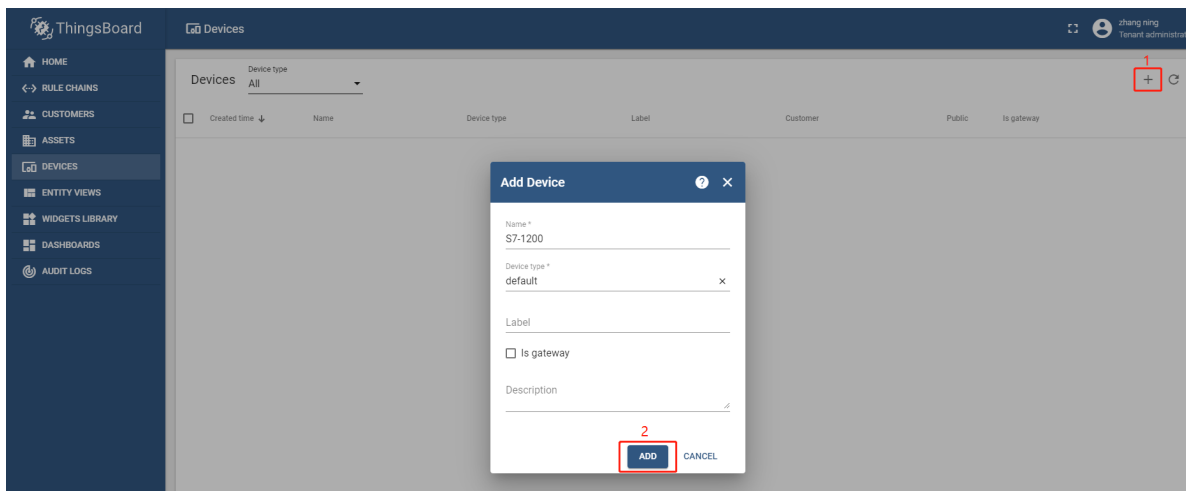


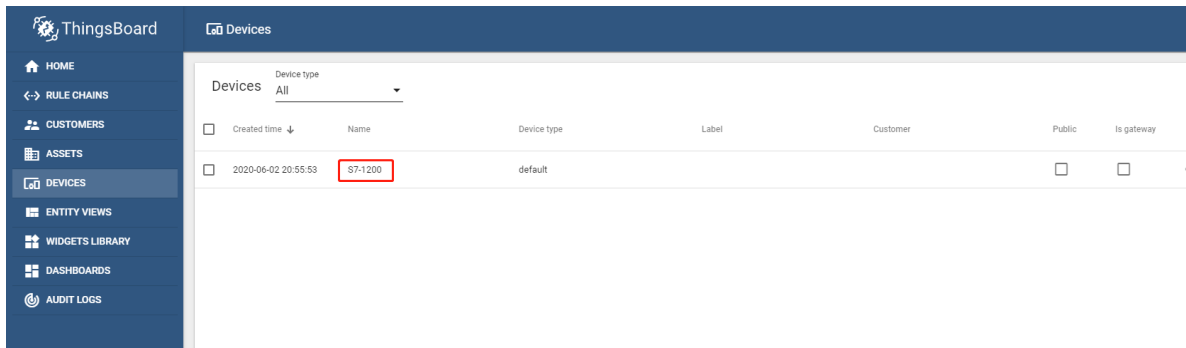


After the asset is added, the page is as follows:

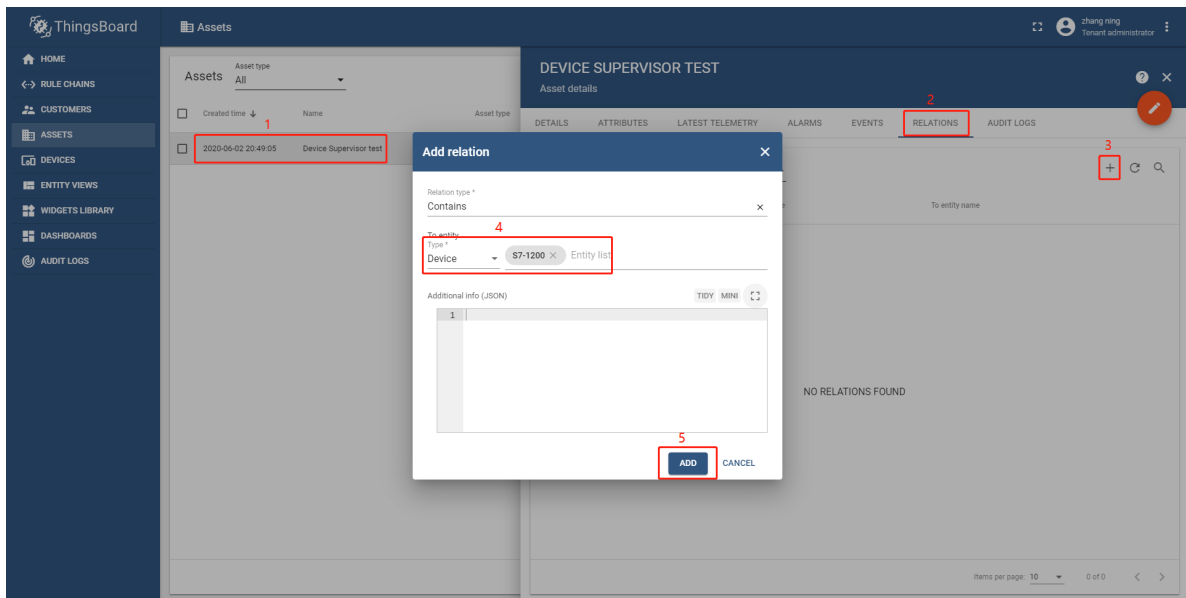


- Add a device

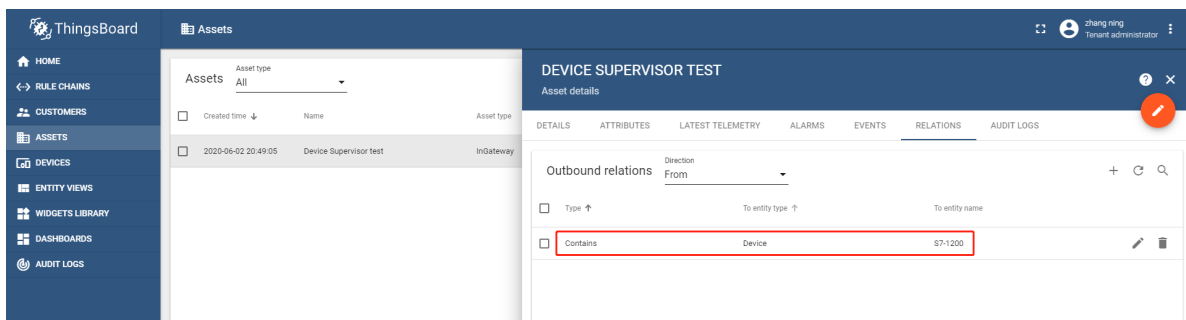




Associate the asset with the device.

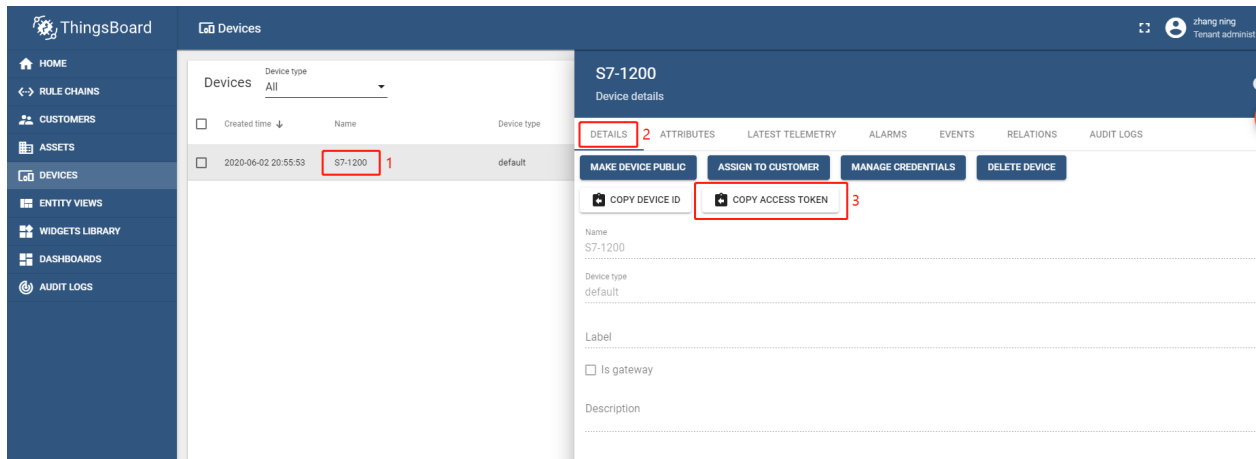


After the device is added, the page is as follows:

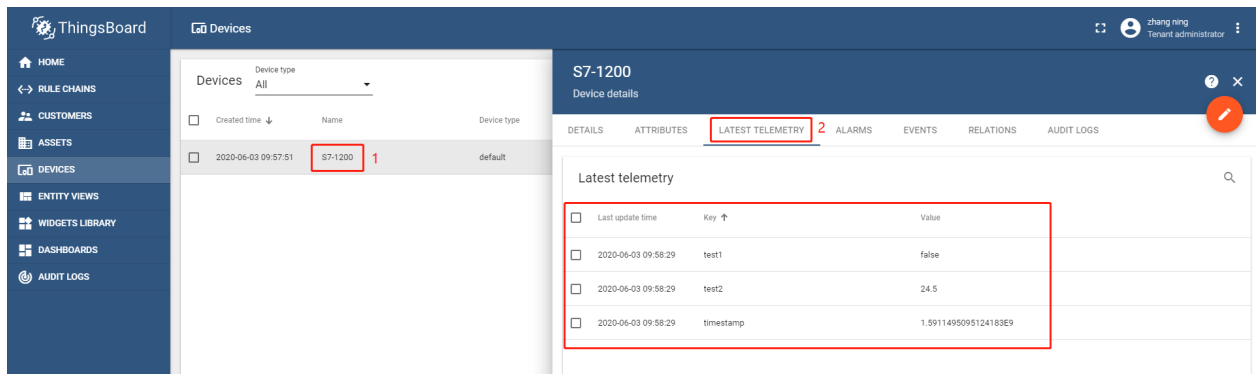


## Transmit the PLC data to ThingsBoard devices

After configuring the asset and device, copy the access token of the added device and paste it to the username field on the cloud service page of the gateway, and then transmit the data to the S7-1200 device in ThingsBoard.



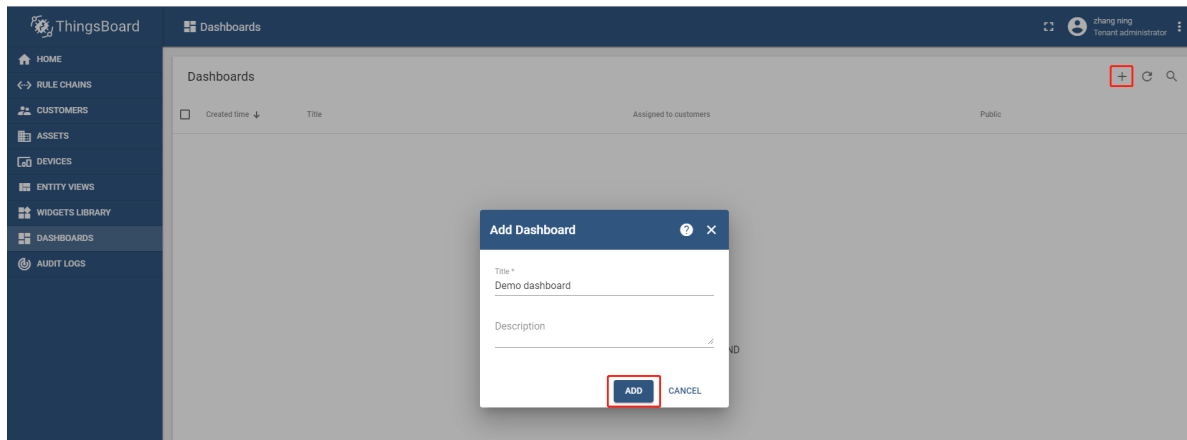
Then, you can view the uploaded data in the latest telemeter of the device.



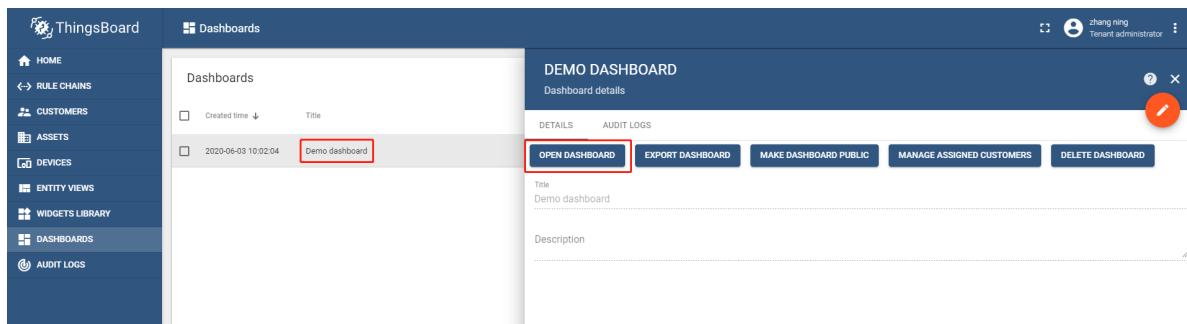
## Configure a visual dashboard

- Add a dashboard
- Add a trend chart
- Add a switch
- Add a dashboard

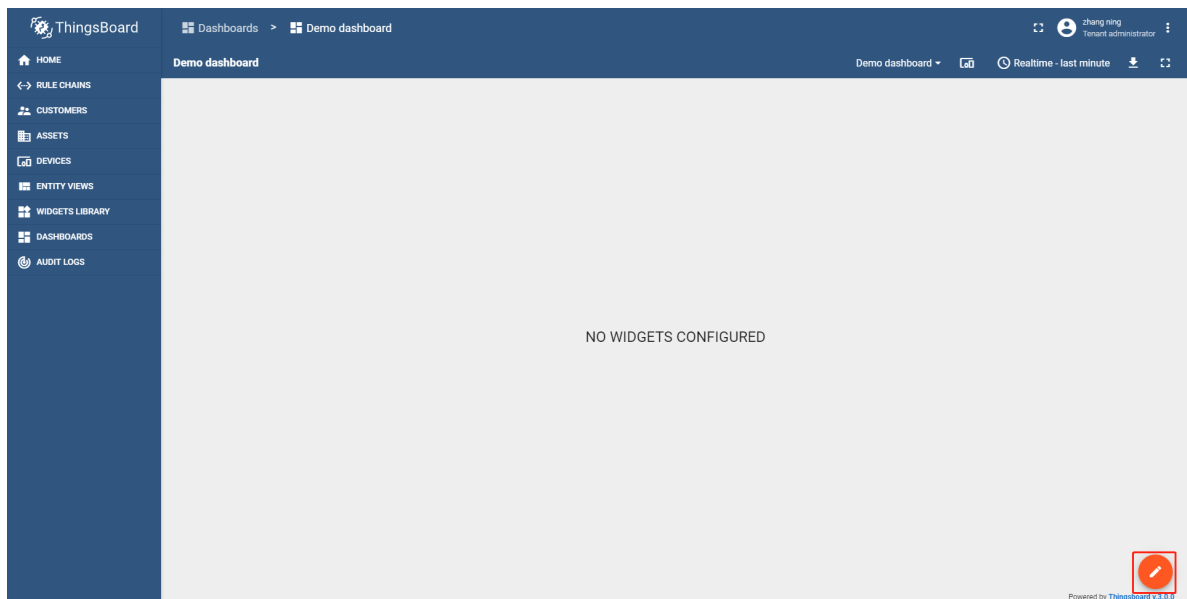
Click **Add Dashboard** and select **Create new dashboard**. On the Add Dashboard page, configure and add a dashboard.



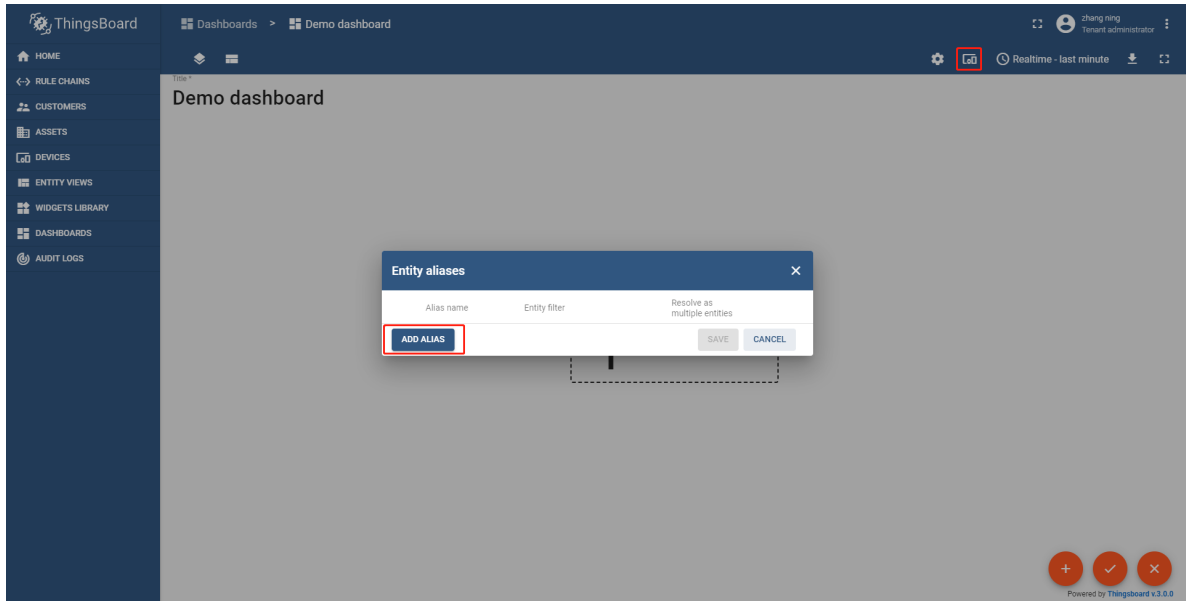
Then, click the dashboard name, and select **OPEN DASHBOARD**.



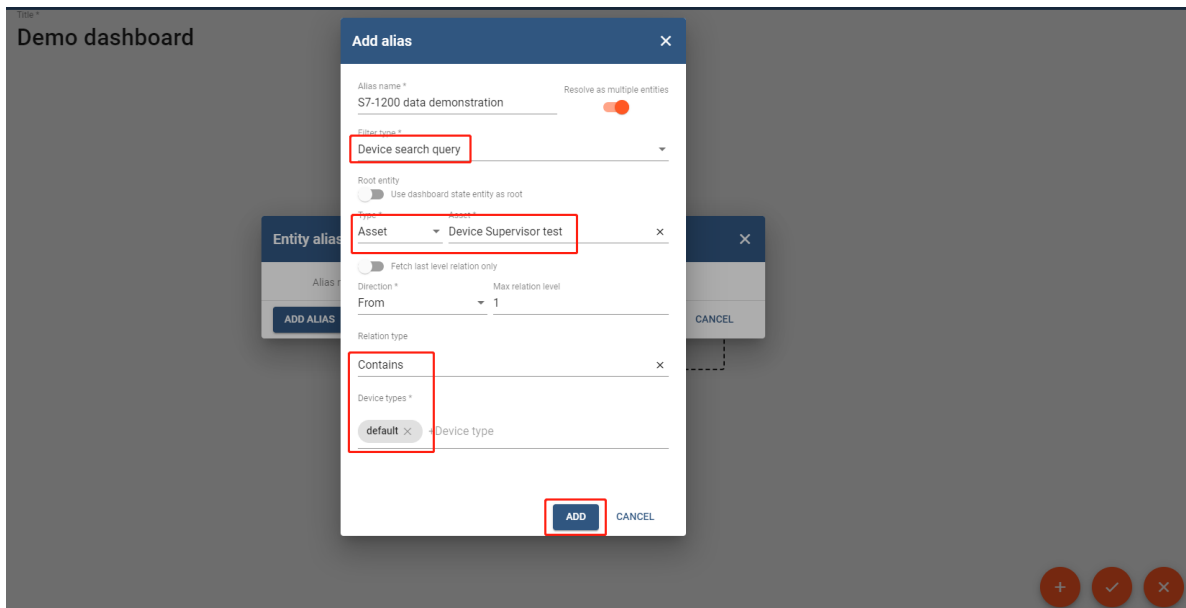
Click **Enter edit mode**.



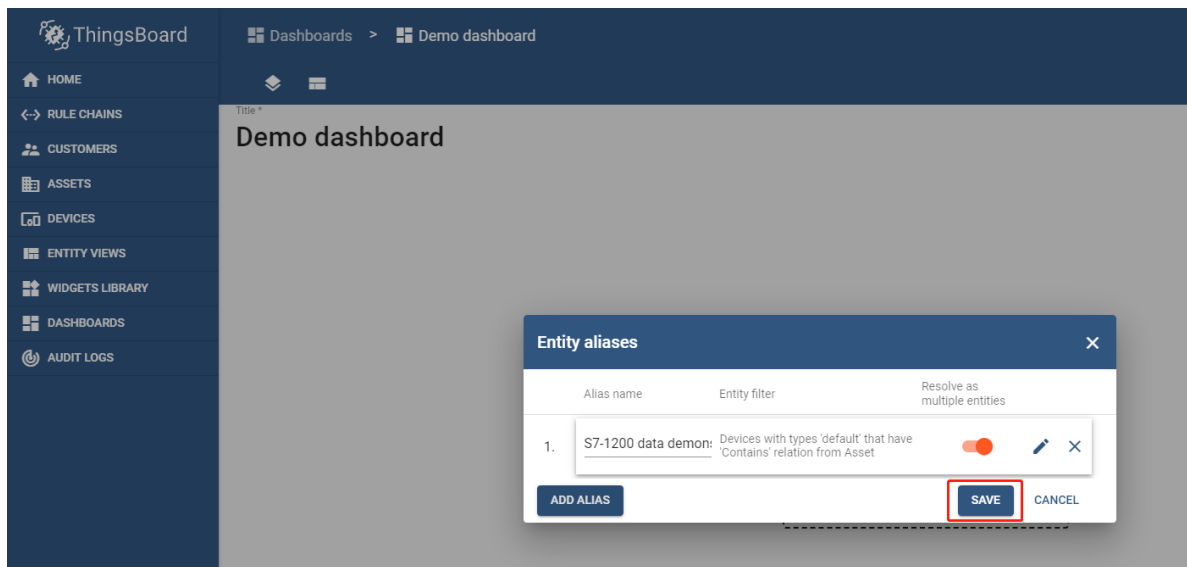
Click **Entity aliases** to add an entity alias for the dashboard.



On the Entity Alias page, configure it according to the following figure:

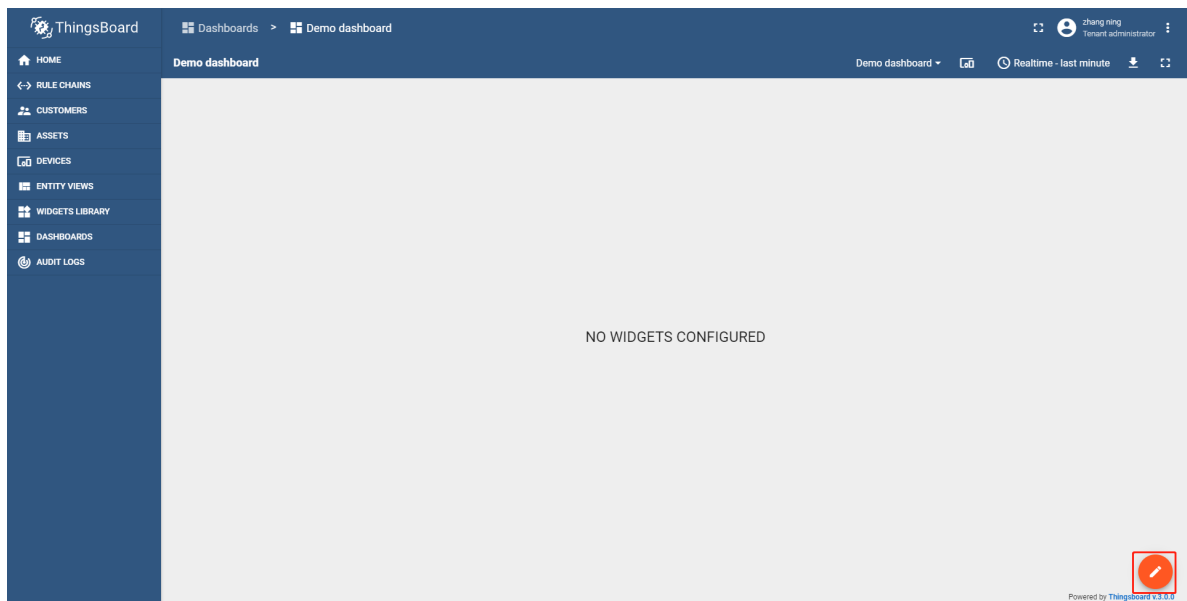


Save it after configuration.



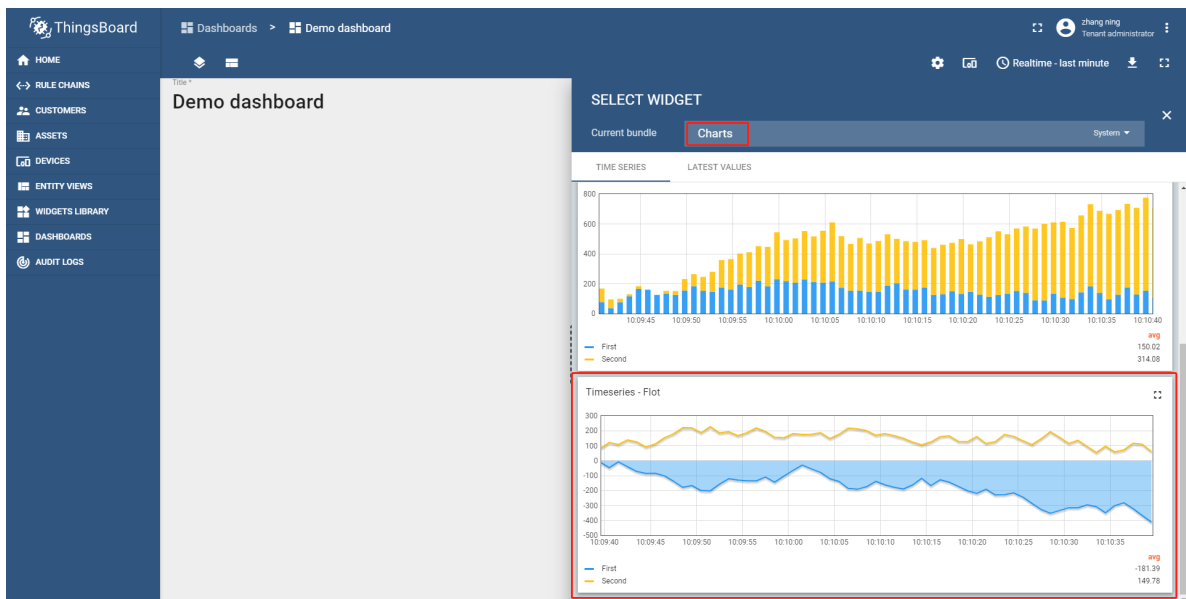
- Add a trend chart

On the dashboard page, click **Enter edit mode** and then click **Add new widget**.

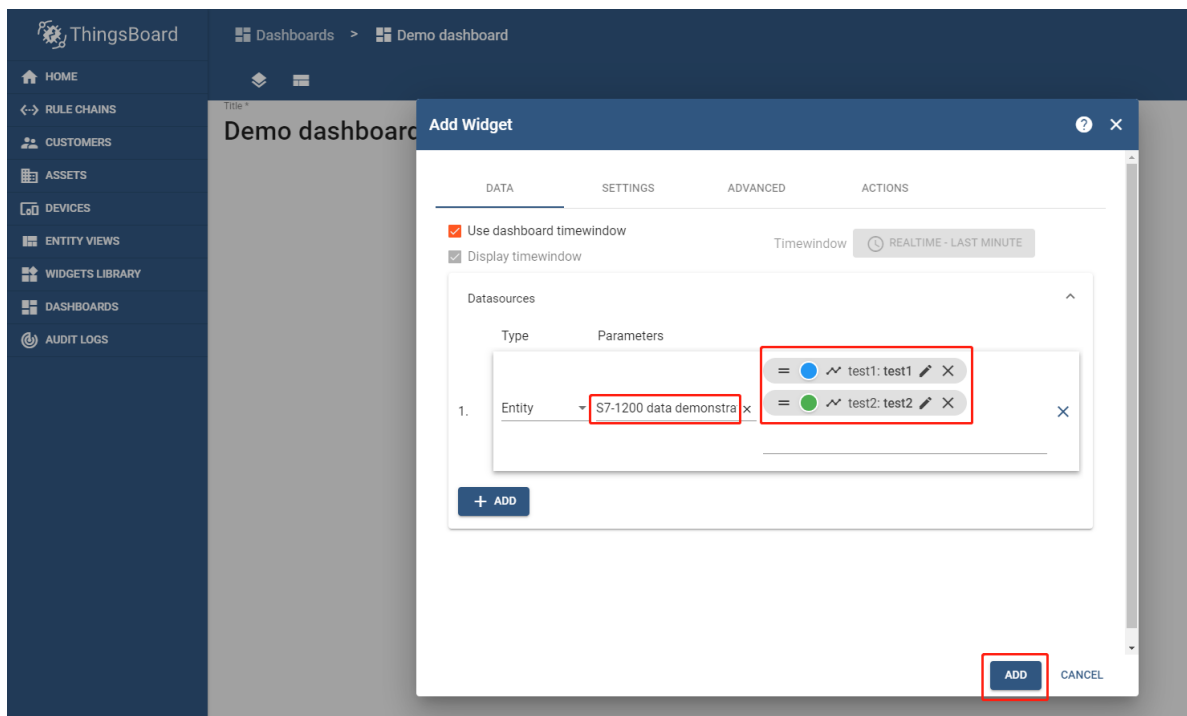
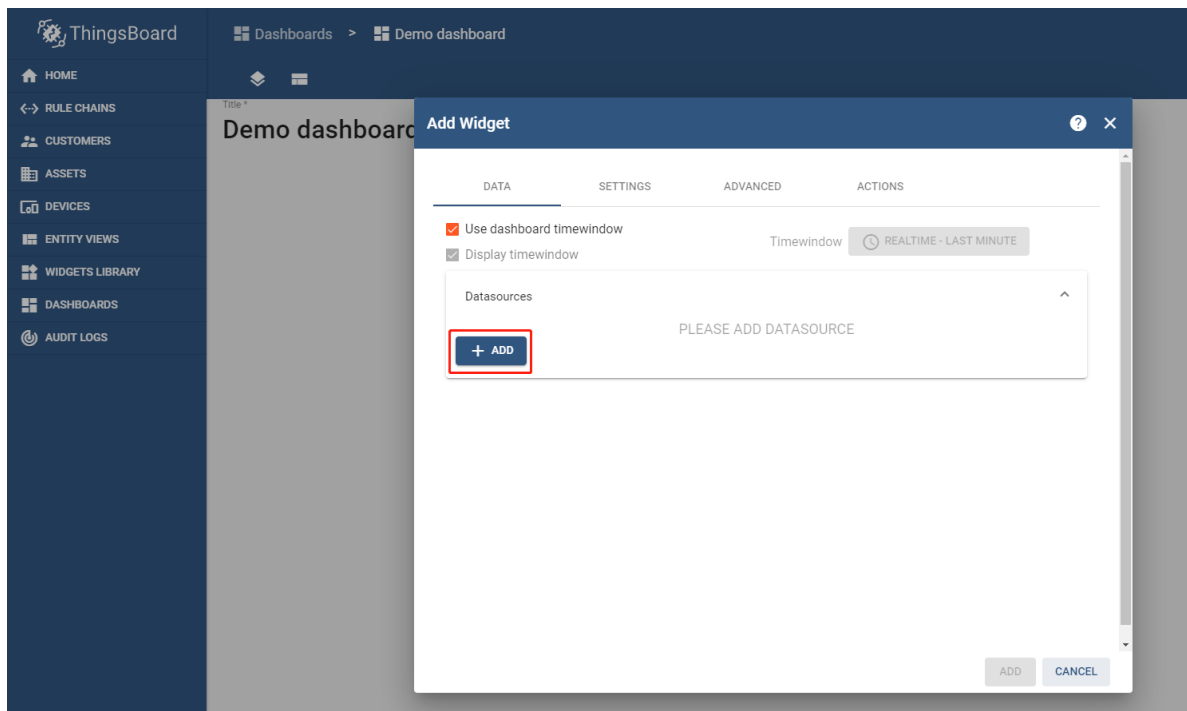




Select **Charts** from components and click **Timeseries-Flot**.

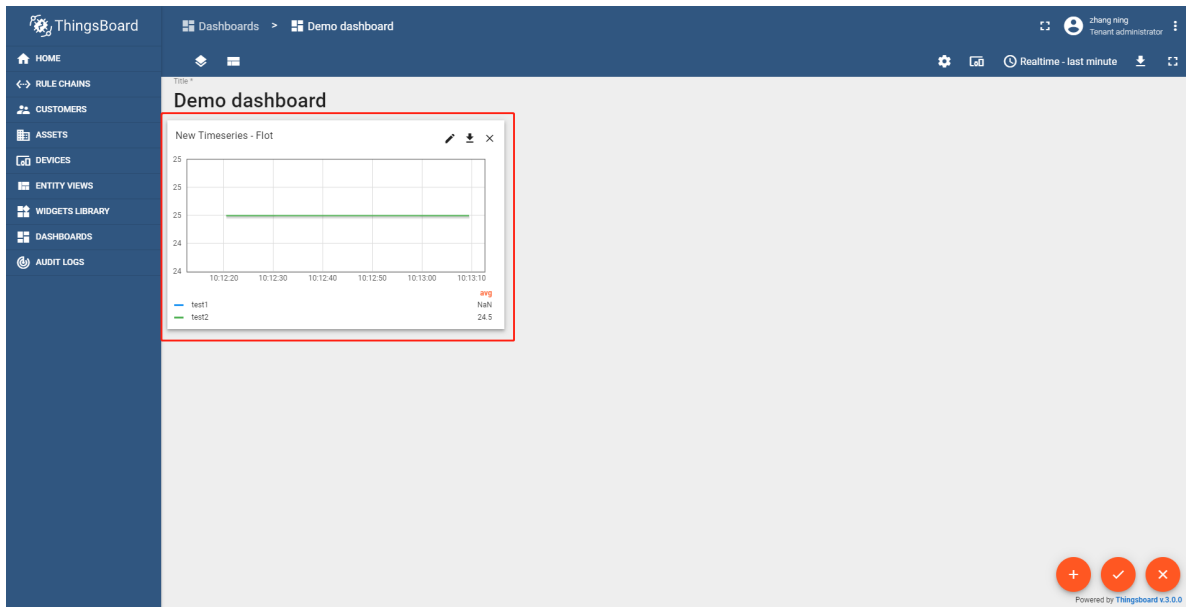


On the **DATA** page of the chart, add data to the trend chart.



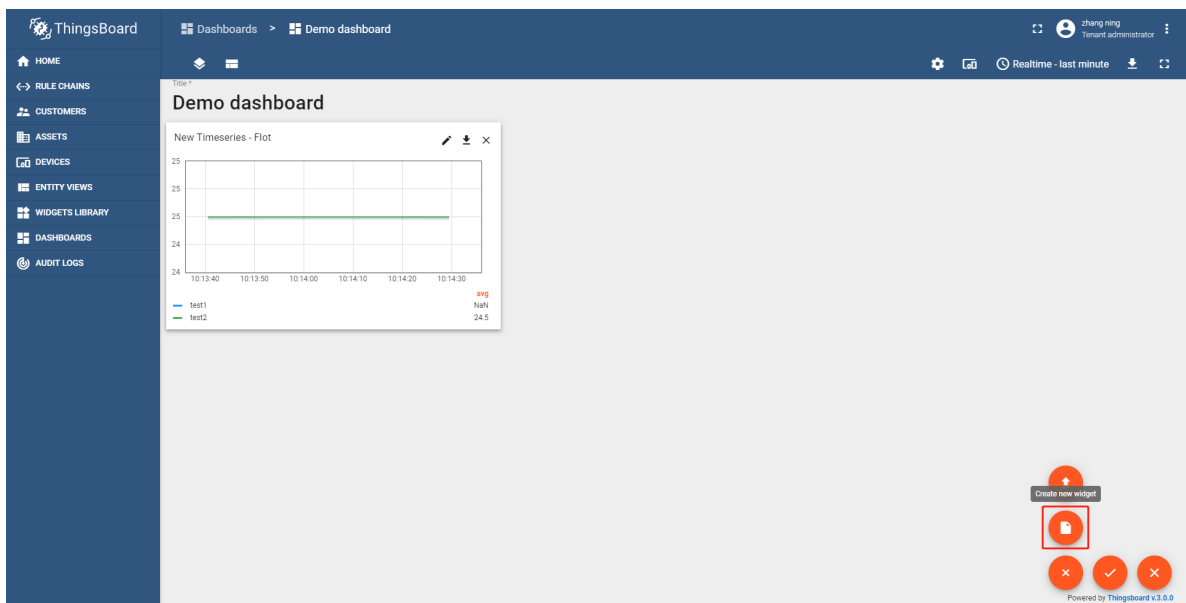
After the data is added, the page is as follows:



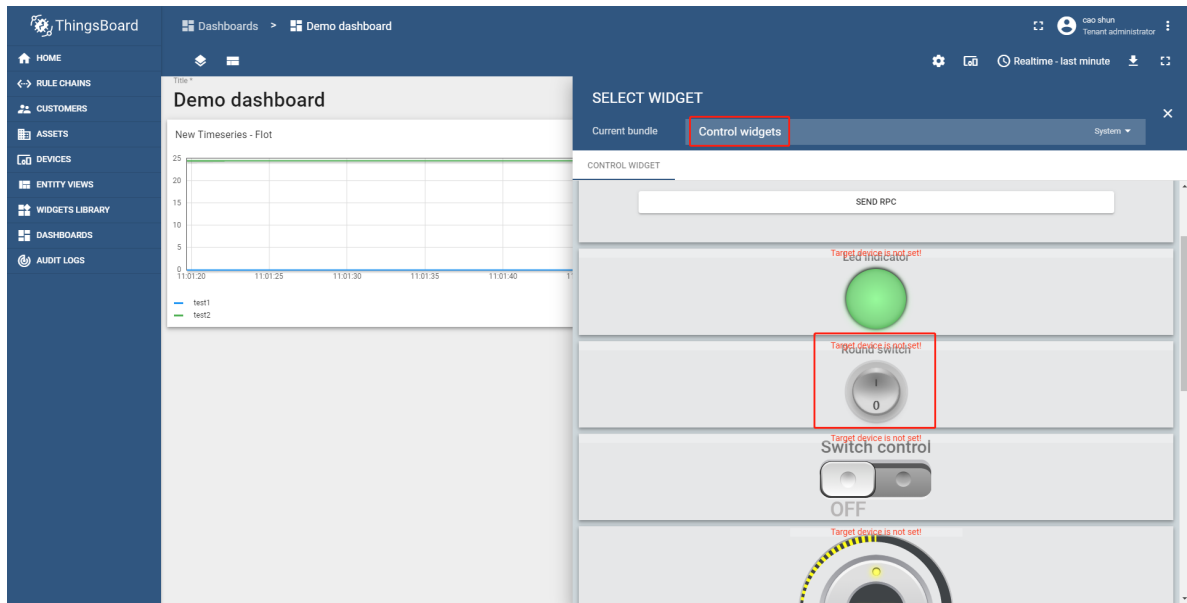


- Add a switch

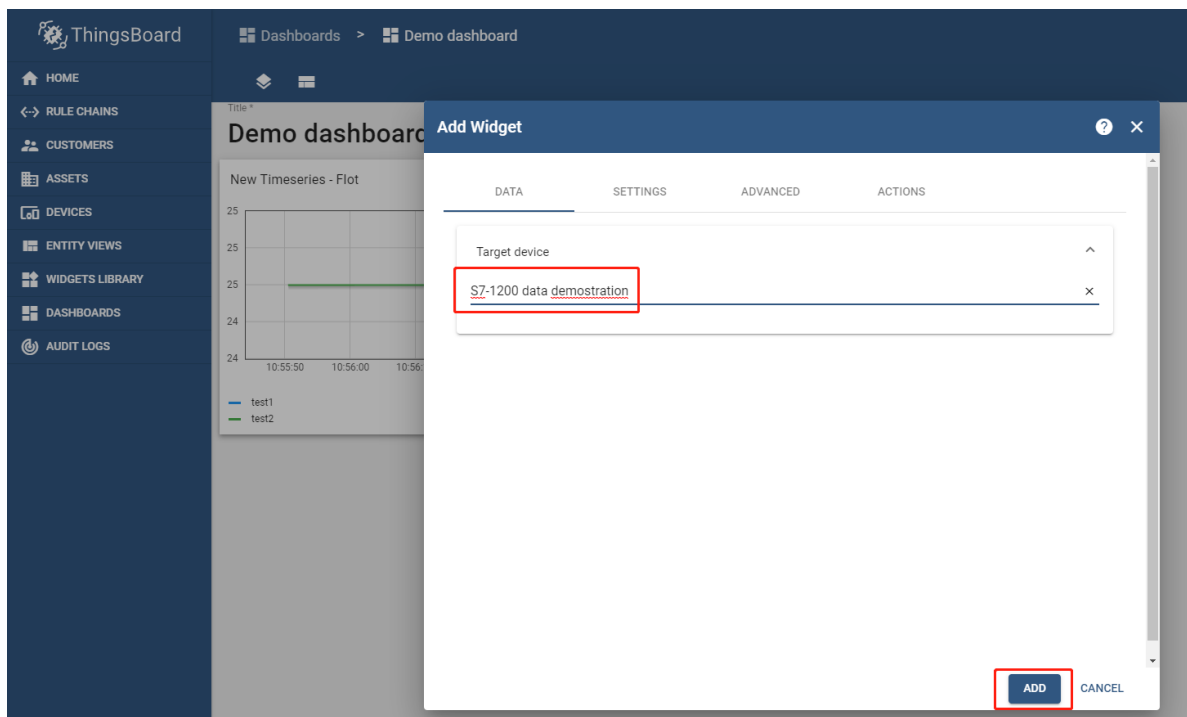
Click **Add new widget**, select **Create new widget**, and add a control switch.



Select **Control widgets** from components, and click **Round switch**.



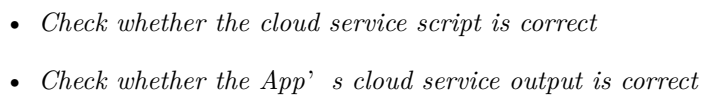
Select the target device.



After configuration, adjust the component size and layout, and save the configuration.



### 1.1.6 FAQ



### Check whether the cloud service script is correct

Open the log of Device Supervisor App Write the script and click **OK**. The Build module: <Main Function Name>, type: <publish/subscribe> information in the log shows whether the script is built successfully.

If the script is built successfully, the page is as follows:

```
[2020-05-18 17:18:27,460] [INFO] [MqttProxy.py 175]: Build module: main, type: publish
[2020-05-18 17:18:27,469] [INFO] [MqttProxy.py 191]: Build OK.
```

If the script failed to build, the page is as follows:

```
[2020-05-18 17:22:04,612] [ERROR] [MqttProxy.py 195]: Build module error. 'No main entry method found. main!'
```

### Check whether the App' s cloud service output is correct

You can use `logger` and `logging` to output important logs. In the following figure, the `logging.info` method is added in the sixth line of the script. You can search for <string> 6 in the log to check whether the output results meet the expectation.

```
[2020-05-18 17:21:53,351] [INFO] [string 6] ['timestamp': '1589793710.2932811', 'values': {'ModbusTest': {'Test1': {'raw_data': False, 'status': 1}, 'Test2': {'raw_data': 31, 'status': 1}, 'Test3': {'raw_data': 0, 'status': 1}}, '57-1200': {'571': {'raw_data': False, 'status': 1}}, 'group_name': 'default']
```

## 1.2 AWS IoT User Manual

The AWS IoT allows for secure bidirectional communication between the AWS cloud and devices (such as sensors, actuators, embedded microcontrollers, and smart devices) connected to the Internet, so that you can collect, store, and analyze telemetering data from devices.

The edge computing gateway InGateway902 (IG902 for short) provides the Device Supervisor app (Device Supervisor for short) to help users connect their devices to the AWS IoT. This document uses IG902 as an example to describe how to submit service data and deliver configuration data between the Device Supervisor and the AWS IoT. For details about the AWS usage restrictions, see [AWS Service Quota](#).

- *Prerequisites*
- *1. Environment Preparation*
  - *1.1 Configuring the AWS IoT*
    - \* *1.1.1 Creating Things*
    - \* *1.1.2 Creating the Policy*
    - \* *1.1.3 Configuring the Certificate*
  - *1.2 Configuring the Edge Computing Gateway*
    - \* *1.2.1 Basic Configuring*
    - \* *1.2.2 Data Collecting Configuration*

- *2. Message Publishing and Subscription*
  - *2.1 Connecting to the AWS IoT*
  - *2.2 Publishing Messages to the AWS IoT*
  - *2.3 Subscribing to AWS IoT Messages*
- *Appendix*
  - *Device Supervisor AWS IoT API Description*

### 1.2.1 Prerequisites

- AWS cloud platform account
- Edge computing gateway IG501/IG902
  - Firmware version
    - \* IG902: IG9-V2.0.0.r12754 or later
    - \* IG501: IG5-V2.0.0.r12884 or later
  - SDK version
    - \* IG902: py3sdk-V1.4.0\_Edge-IG9 or later
    - \* IG501: py3sdk-V1.4.0\_Edge-IG5 or later
  - App version: device\_supervisor-V1.2.5 or later

### 1.2.2 1. Environment Preparation

- *1.1 Configuring the AWS IoT*
- *1.2 Configuring the Edge Computing Gateway*

#### 1.1 Configuring the AWS IoT

- *1.1.1 Creating Objects*
- *1.1.2 Creating the Policy*
- *1.1.3 Configuring the Certificate*

If you have configured the things, policy, and certificate in the AWS IoT console, go to *1.2 Configuring the Edge Computing Gateway*. Otherwise, perform the following steps to configure the AWS IoT console. Visit <https://aws.amazon.com/>, log in to the IoT console, and choose **IoT Core**.

The screenshot shows the AWS website interface. At the top is the AWS logo and navigation links: Products, Solutions, Pricing, Documentation, Learn, Partner Network, AWS Marketplace, Customer Enablement, Events, Explore More, and a search icon. On the right, there are links for Contact Sales, Support, English, My Account, and a prominent orange button labeled "Sign In to the Console". Below the navigation bar is a purple banner with the text "See the AWS Initiatives and Response to COVID-19 >".

The main content area features a large banner for "Amazon Elasticsearch Service" with the tagline "Fully managed Elasticsearch for log analytics, without the operational overhead" and a "Learn more >" link. To the right of the banner is an illustration of a rocket launching from a computer screen.

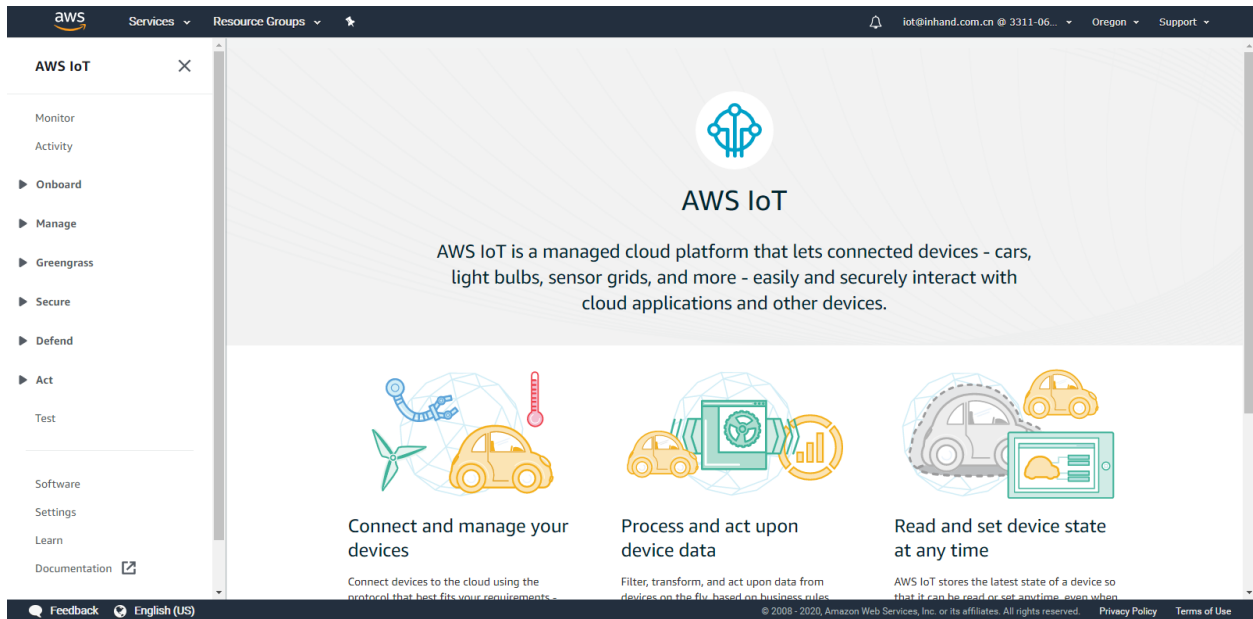
Below the banner is a row of four service tiles:
 

- Amazon Lightsail**: Everything you need to get started on AWS.
- Your Data Has Hidden Value**: Learn how to get the most from your data.
- Get Hands-On with SageMaker**: Learn how to build, train, and deploy a machine learning model.
- Amazon Aurora**: Performance and availability of MySQL and PostgreSQL.

The bottom section displays a grid of AWS service categories, each with an icon and a list of services:
 

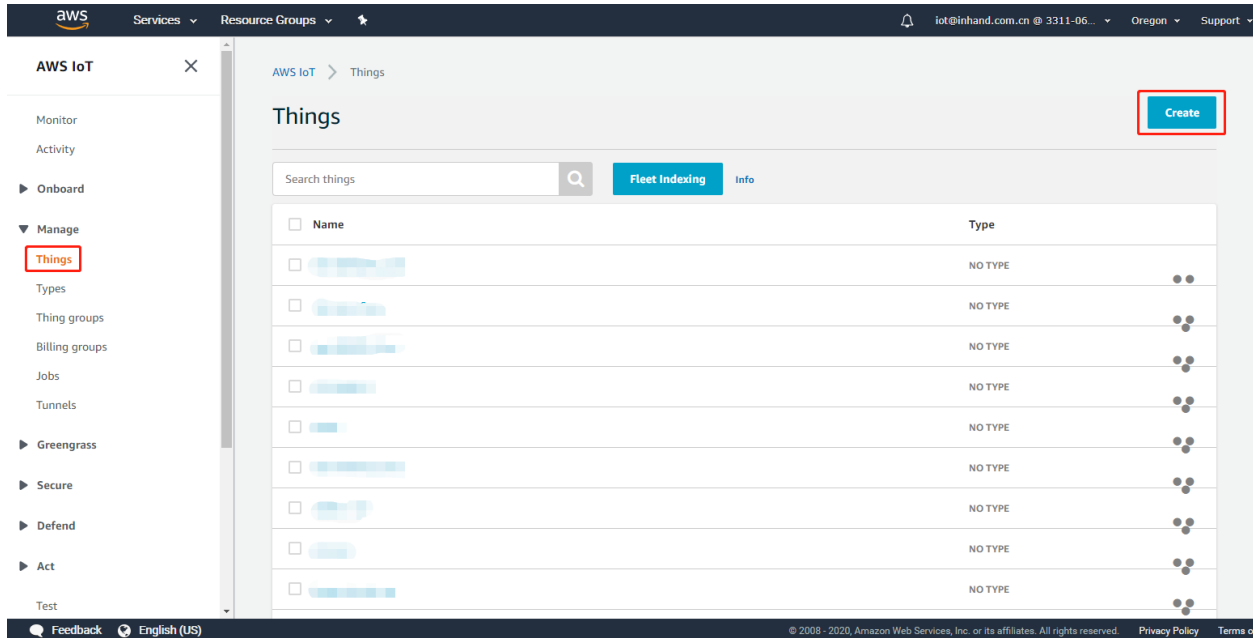
- Networking & Content Delivery**: VPC, CloudFront, Route 53, API Gateway, Direct Connect, AWS App Mesh, AWS Cloud Map, Global Accelerator.
- Developer Tools**: CodeStar, CodeCommit, CodeArtifact, CodeBuild, CodeDeploy, CodePipeline, Cloud9, X-Ray.
- Customer Enablement**: AWS IQ, Support, Managed Services.
- Robotics**: AWS RoboMaker.
- Blockchain**: Amazon Managed Blockchain.
- Amazon Compute**: Amazon Forecast, Amazon Fraud Detector, Amazon Kendra, Amazon Lex, Amazon Personalize, Amazon Polly, Amazon Rekognition, Amazon Textract, Amazon Transcribe, Amazon Translate, AWS DeepComposer, AWS DeepLens, AWS DeepRacer.
- Analytics**: Athena, EMR, CloudSearch, Elasticsearch Service, Kinesis, QuickSight, Data Pipeline, AWS Data Exchange, AWS Glue, AWS Lake Formation, MSK.
- Customer Engagement**: Amazon Connect, Pinpoint, Simple Email Service.
- Business Applications**: Alexa for Business, Amazon Chime, WorkMail, Amazon Honeycode.
- End User Computing**: WorkSpaces, AppStream 2.0, WorkDocs, WorkLink.
- Internet of Things**: **IoT Core** (highlighted with a red box), FreeRTOS, IoT 1-Click, IoT Analytics, IoT Device Defender, IoT Device Management, IoT Events, IoT Greengrass, IoT SiteWise, IoT Things Graph.
- Game Development**: (category icon shown).

The following page appears after you log in to **IoT Core**:



### 1.1.1 Creating Things

Choose **Manage > Things** and click **Create**.



Click **Create** a single thing.

## Creating AWS IoT things

An IoT thing is a representation and record of your physical device in the cloud. Any physical device needs a thing record in order to work with AWS IoT. [Learn more.](#)

### Register a single AWS IoT thing

Create a thing in your registry

Create a single thing

### Bulk register many AWS IoT things

Create things in your registry for a large number of devices already using AWS IoT, or register devices so they are ready to connect to AWS IoT.

Create many things

Cancel

Create a single thing

Set the thing name, for example, `aws_iot_test`, retain the default values for other parameters, and then click **Next**.



CREATE A THING

Add your device to the thing registry

STEP  
1/3

This step creates an entry in the thing registry and a thing shadow for your device.

Name

aws\_iot\_test

Apply a type to this thing

Using a thing type simplifies device management by providing consistent registry data for things that share a type. Types provide things with a common set of attributes, which describe the identity and capabilities of your device, and a description.

Thing Type

No type selected

Create a type

Add this thing to a group

Adding your thing to a group allows you to manage devices remotely using jobs.

Thing Group

Groups /

Create group Change

Set searchable thing attributes (optional)

Enter a value for one or more of these attributes so that you can search for your things in the registry.

Attribute key	Value	
Provide an attribute key, e.g. Manufactur	Provide an attribute value, e.g. Acme-Cor	Clear
<div>Add another</div>		

Show thing shadow ▾

Cancel

Back

Next

Click **Create certificate**.

CREATE A THING

STEP 2/3

Add a certificate for your thing

A certificate is used to authenticate your device's connection to AWS IoT.

One-click certificate creation (recommended)

This will generate a certificate, public key, and private key using AWS IoT's certificate authority.

Create certificate

Create with CSR

Upload your own certificate signing request (CSR) based on a private key you own.

Create with CSR

Use my certificate

Register your CA certificate and use your own certificates for one or many devices.

Get started

Skip certificate and create thing

You will need to add a certificate to your thing later before your device can connect to AWS IoT.

Create thing without certificate

After the certificate is created, you need to download the certificate for this thing, the private key, and the root CA of the AWS IoT, activate the certificate, and then click **Done**. It is recommended that you download **Amazon Root CA 1** or **Starfield Starfield root CA certificate** when downloading the root CA certificate. Currently, the Amazon Root CA 3 certificate is not supported.

98

Chapter 1. InGateway Documentation Site Navigation

## Certificate created!

Download these files and save them in a safe place. Certificates can be retrieved at any time, but the private and public keys cannot be retrieved after you close this page.

In order to connect a device, you need to download the following:

A certificate for this thing	82a0af6572.cert.pem	<a href="#">Download</a>
A public key	82a0af6572.public.key	<a href="#">Download</a>
A private key	82a0af6572.private.key	<a href="#">Download</a>

You also need to download a root CA for AWS IoT:

A root CA for AWS IoT [Download](#)

[Activate](#)

[Cancel](#)

[Done](#)

[Attach a policy](#)

[AWS](#) > [Documentation](#) > [AWS IoT](#) > [Developer Guide](#)

### AWS IoT

Developer Guide

- What is AWS IoT?
- Getting started with AWS IoT Core
- AWS IoT Tutorials
- Managing devices with AWS IoT
- Tagging your AWS IoT resources
- Security
  - Security in AWS IoT
    - Authentication
      - Server authentication**
        - Client authentication
        - Custom authentication

## CA certificates for server authentication

Depending on which type of data endpoint you are using and which cipher suite you have negotiated, AWS IoT Core server authentication certificates are signed by one of the following root CA certificates:

### VeriSign Endpoints (legacy)

- RSA 2048 bit key: [VeriSign Class 3 Public Primary G5 root CA certificate](#)

### Amazon Trust Services Endpoints (preferred)

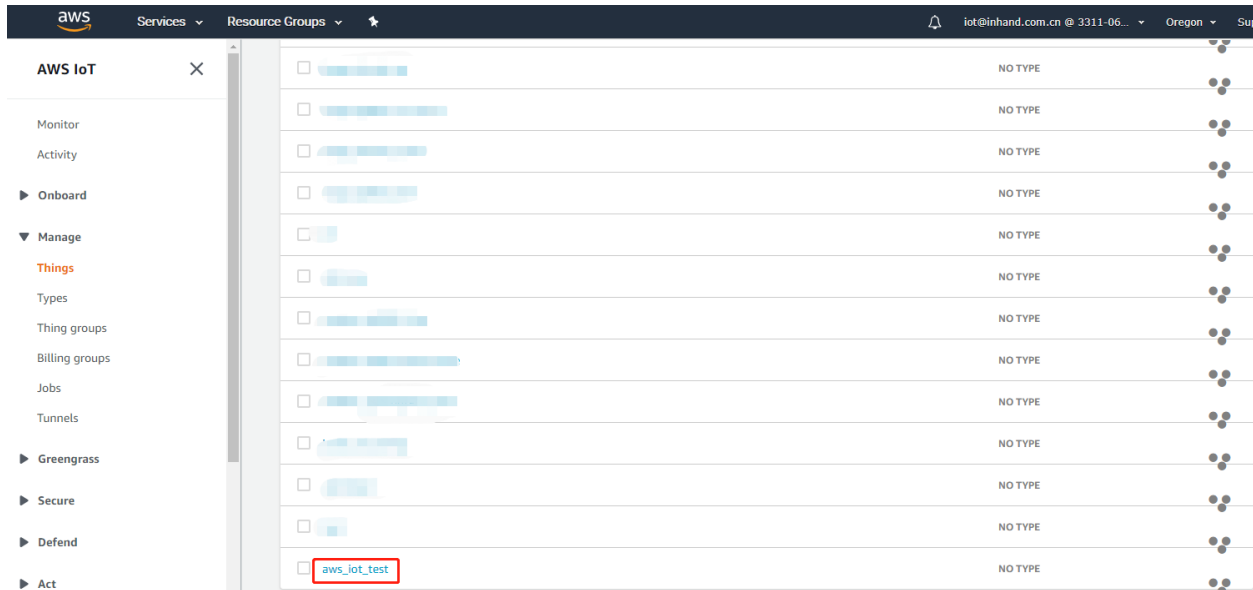
**Note**

You might need to right click these links and select **Save link as...** to save these certificates as files.

- RSA 2048 bit key: [Amazon Root CA 1](#)
- RSA 4096 bit key: Amazon Root CA 2. Reserved for future use.
- ECC 256 bit key: [Amazon Root CA 3](#)
- ECC 384 bit key: Amazon Root CA 4. Reserved for future use.

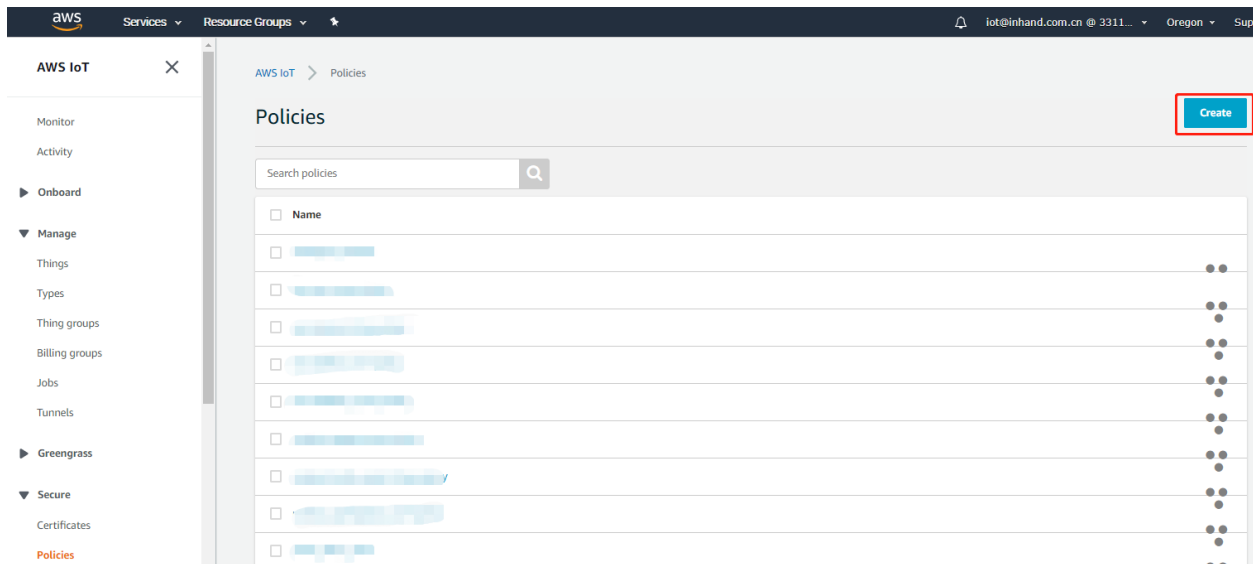
These certificates are all cross-signed by the [Starfield Root CA Certificate](#). All new AWS IoT Core regions, beginning with the May 9, 2018 launch of AWS IoT Core in the Asia Pacific (Mumbai) Region, serve only ATS certificates.

After the thing is created, the following page is displayed:



### 1.1.2 Creating the Policy

Choose **Secure > Policies** and click **Create**.



On the **Create a policy** page, enter the policy name, configure the policy by referring to the following settings, and then click **Create**. This policy allows all clients to connect to the AWS IoT.

- Enter `iot:*` in the Action text box.
- Enter `*` in the Resource ARN text box.
- Select **Allow** for **Effect**.

## Create a policy

Create a policy to define a set of authorized actions. You can authorize actions on one or more resources (things, topics, topic filters). To learn more about IoT policies go to the [AWS IoT Policies documentation page](#).

Name

aws\_iot\_test

### Add statements

Policy statements define the types of actions that can be performed by a resource. Advanced mode

Action

iot:\*

Resource ARN

\*

Effect

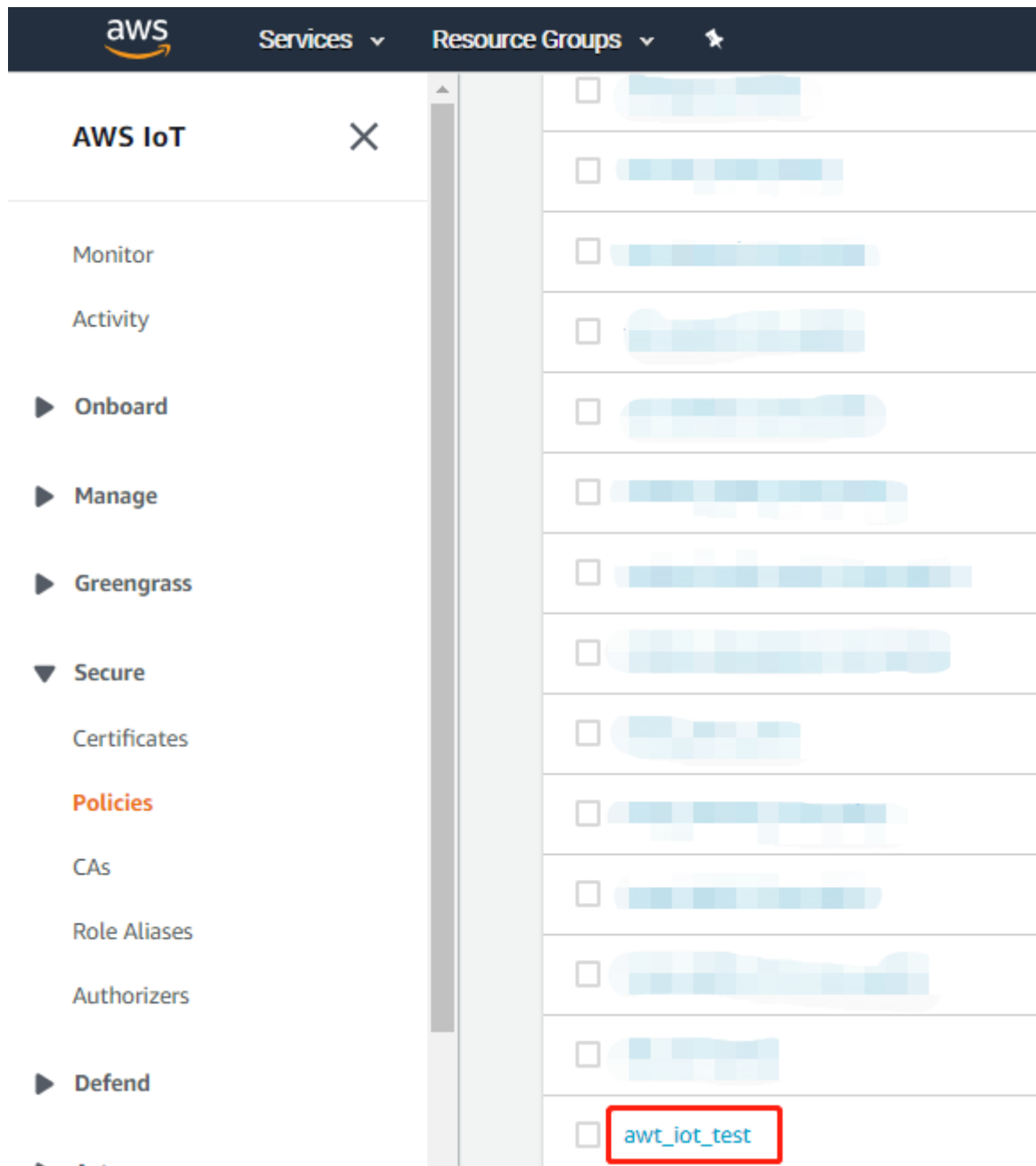
☒ Allow ☐ Deny

Remove

Add statement

Create

After the policy is created, the following page is displayed:



### 1.1.3 Configuring the Certificate

Choose **Secure** > **Certificates**. The following page is displayed:

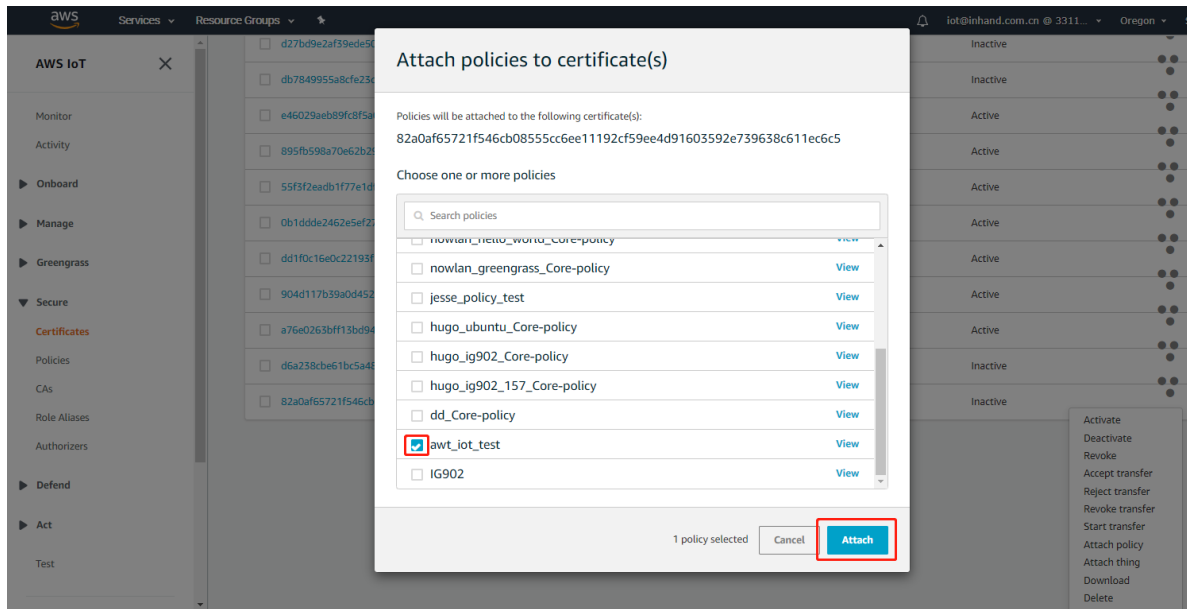
AWS IoT	<input type="checkbox"/> ce08826dfa010e0756a77e62ff5c90b69d4e9bf524aec312b932f1a162eae69b Active
	<input type="checkbox"/> 14c90745c8be0aa84f99fada344be270ad934600bed0a96f3857111aa7fce426 Inactive
Monitor	<input type="checkbox"/> 1dc5c64b0eed901d51d7e0e976fd610a097b9e454b25d63154e078d6987a7dd3 Active
Activity	<input type="checkbox"/> bf4772c7abc964d1cff313c22494ae39c17b0f3fb72b65a8b655834cded461c Inactive
▶ Onboard	<input type="checkbox"/> d27bd9e2af39ede5096af2ea882ee904c47a4f3c3f5d299dad1a6460031ab2b7 Inactive
▶ Manage	<input type="checkbox"/> db7849955a8cfe23ce6c606744e3b5a47352c5bdfa4261fac3989bf49110c3f Inactive
▶ Greengrass	<input type="checkbox"/> e46029aeb89fc8f5a03075ab01ee82add94ff01c7c53ddf8515fbd17e2711775 Active
▼ Secure	<input type="checkbox"/> 895fb598a70e62b29f981e59cef91e6fce40c42056d5bee2778ceabc3e79593 Active
Certificates	<input type="checkbox"/> 55f3f2eadb1f77e1df0b7166ca86290638db9659031963859f02ae6589d10bc Active
Policies	<input type="checkbox"/> 0b1ddde2462e5ef278899b05781c493524cdef764379fd38b2f1bc9d5fb099be Active
CAs	<input type="checkbox"/> dd1f0c16e0c22193f1b2312bef35adee3c9735749a2925e8b086c24f00042bf9 Active
Role Aliases	<input type="checkbox"/> 904d117b39a0d452be3756a2052a30b8490bcb418c557b8750e84d45d9a108a6 Active
Authorizers	<input type="checkbox"/> a76e0263bff13bd94694b45bd93b3ea65a35495d6ae38d5da9353d231283cdcb Active
▶ Defend	<input type="checkbox"/> d6a238cbe61bc5a48868daa9ca96fcd058a79a093e779da8e4d5c4e88f25d2b9 Inactive
▶ Act	<input type="checkbox"/> 82a0af65721f546cb08555cc6ee11192cf59ee4d91603592e739638c611ec6c5 Inactive

- Attaching a policy

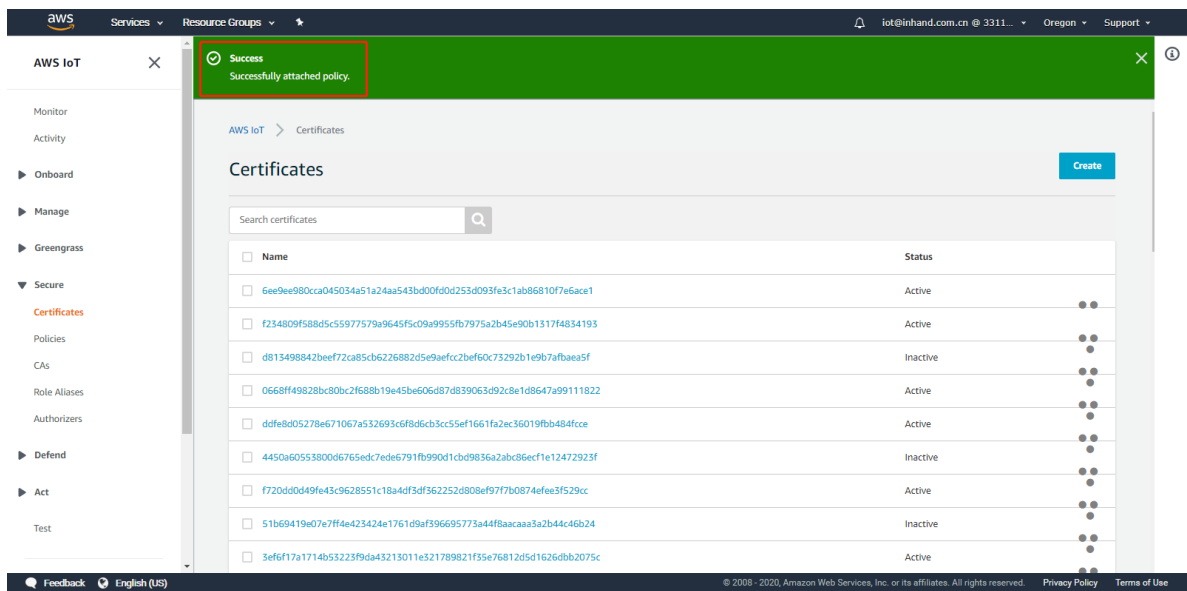
Click ... to the right of the certificate and choose **Attach policy** to attach a policy to the certificate.

AWS IoT	<input type="checkbox"/> d27bd9e2af39ede5096af2ea882ee904c47a4f3c3f5d299dad1a6460031ab2b7 Inactive
	<input type="checkbox"/> db7849955a8cfe23ce6c606744e3b5a47352c5bdfa4261fac3989bf49110c3f Inactive
Monitor	<input type="checkbox"/> e46029aeb89fc8f5a03075ab01ee82add94ff01c7c53ddf8515fbd17e2711775 Active
Activity	<input type="checkbox"/> 895fb598a70e62b29f981e59cef91e6fce40c42056d5bee2778ceabc3e79593 Active
▶ Onboard	<input type="checkbox"/> 55f3f2eadb1f77e1df0b7166ca86290638db9659031963859f02ae6589d10bc Active
▶ Manage	<input type="checkbox"/> 0b1ddde2462e5ef278899b05781c493524cdef764379fd38b2f1bc9d5fb099be Active
▶ Greengrass	<input type="checkbox"/> dd1f0c16e0c22193f1b2312bef35adee3c9735749a2925e8b086c24f00042bf9 Active
▼ Secure	<input type="checkbox"/> 904d117b39a0d452be3756a2052a30b8490bcb418c557b8750e84d45d9a108a6 Active
Certificates	<input type="checkbox"/> a76e0263bff13bd94694b45bd93b3ea65a35495d6ae38d5da9353d231283cdcb Active
Policies	<input type="checkbox"/> d6a238cbe61bc5a48868daa9ca96fcd058a79a093e779da8e4d5c4e88f25d2b9 Inactive
CAs	<input type="checkbox"/> 82a0af65721f546cb08555cc6ee11192cf59ee4d91603592e739638c611ec6c5 Inactive
Role Aliases	
Authorizers	
▶ Defend	
▶ Act	
Test	

Select the policy that you have created, and click **Attach**.



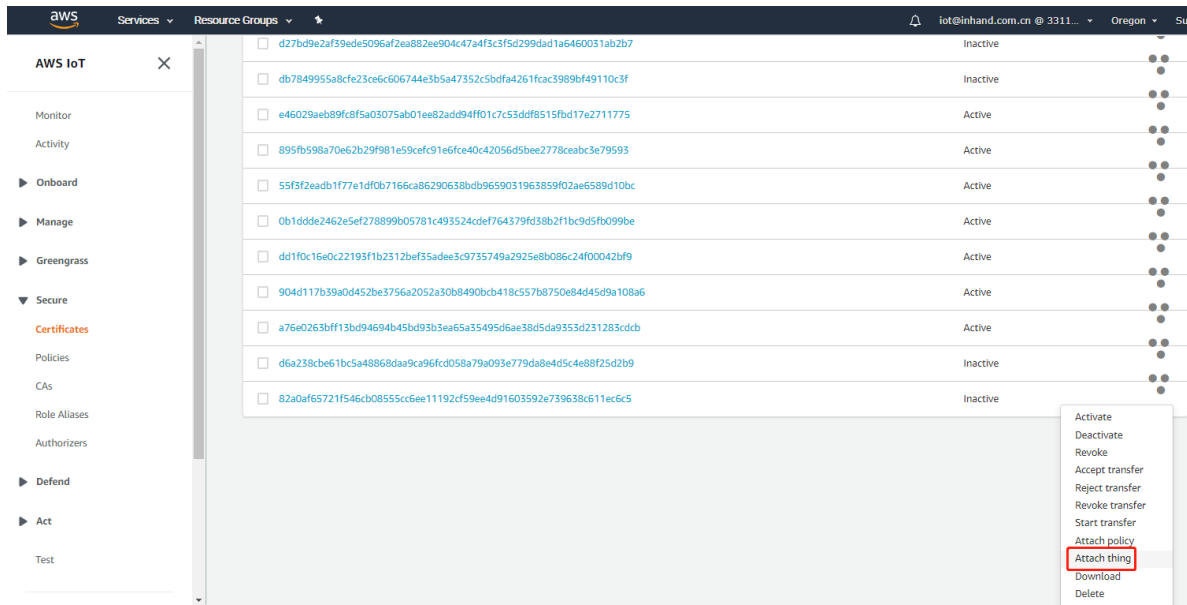
After the policy is attached, the following page is displayed:



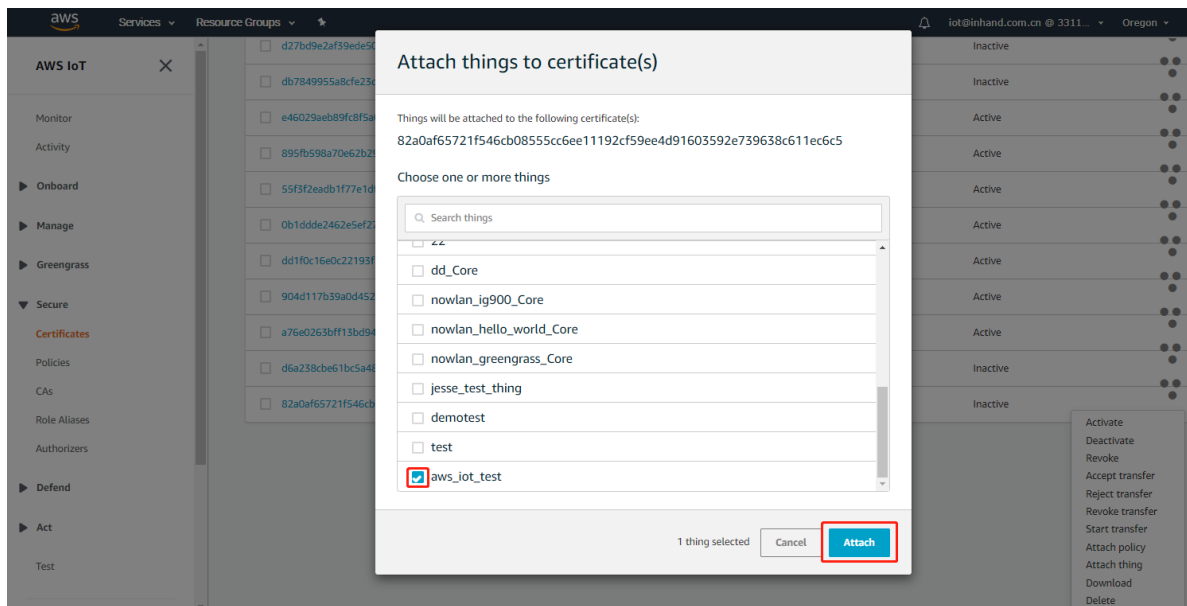
- Attaching a thing

Click **...** to the right of the certificate and choose **Attach thing** to attach a thing to the certificate.

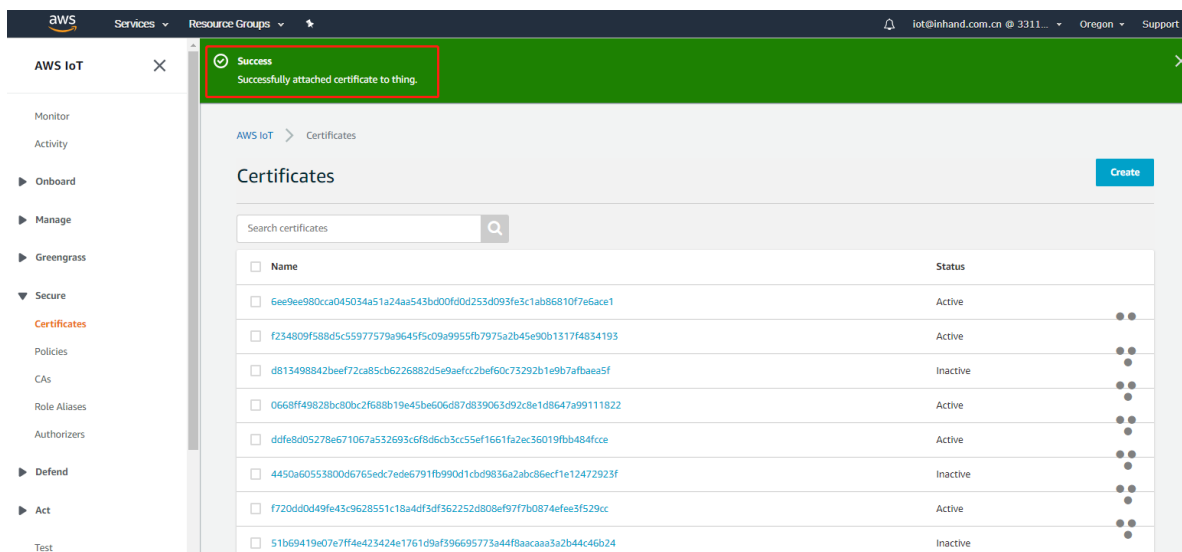




Select the thing that you have created, and click **Attach**.



After the thing is attached, the following page is displayed:



The preparation of the AWS IoT environment is completed.

## 1.2 Configuring the Edge Computing Gateway

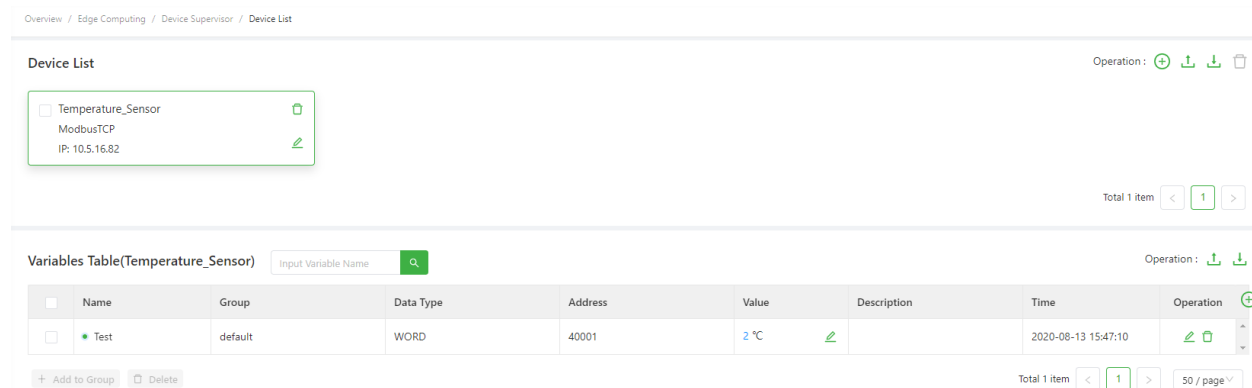
- 1.2.1 Basic Configuring
- 1.2.2 Data Collecting Configuration

### 1.2.1 Basic Configuring

- For details about IG902 connection configuration and software version update, see [IG902 Quick Guide](#).
- For details about IG501 connection configuration and software version update, see [IG501 Quick Guide](#).

### 1.2.2 Data Collecting Configuration

For details about the basic data collection configuration for the Device Supervisor, see [Device Supervisor App User Manual](#). The following figure shows the data collection configuration in this document:



## 1.2.3 2. Message Publishing and Subscription

- *2.1 Connecting to the AWS IoT*
- *2.2 Publishing Messages to the AWS IoT*
- *2.3 Subscribing to AWS IoT Messages*

The topics that start with a dollar sign (\$) are reserved for the AWS IoT. You can subscribe to and publish messages to these topics. However, you cannot create topics with a “\$” prefix. Prohibited message publishing or subscription for the reserved topics may cause connection failures. For details about the topics reserved for the AWS IoT, see [Reserved Topics](#).

### 2.1 Connecting to the AWS IoT

Choose **Edge Computing > Device Supervisor > Cloud** on IG902, select **Enable Cloud Service**, and select **AWS IoT** from the Type drop-down list. The following is a configuration example:

[Overview](#) / [Edge Computing](#) / [Device Supervisor](#) / [Cloud](#)

---

## Status

Cloud Status: Connection Successful

Connection time: 0 Day 00:00:48

---

### Enable Cloud Service:



\* Type:

AWS IoT

\* Endpoint:

-----ats.iot.us-wes

\* Client ID:

awstest


\* Certificate For Thing:

82a0af6572-certificate.pem.crt

 Import


\* Private Key:

82a0af6572-private.pem.key

 Import

\* rootCA:

AmazonRootCA1.pem

 Import

[Advanced Settings](#) >

**Submit**

Reset

The parameters are described as follows:

- **Type:** select **AWS IoT** for an AWS IoT connection.
- **Endpoint:** endpoint address of the AWS IoT, which can be obtained from the **Settings** page of the AWS IoT. If the VeriSign Class 3 Public Primary G5 root CA certificate is used, you need to delete “-ats” from the address.

**Settings**

**Custom endpoint** ENABLED

This is your custom endpoint that allows you to connect to AWS IoT. Each of your Things has a REST API available at this endpoint. This is also an important property to insert when using an MQTT client or the AWS IoT [Device SDK](#).

Your endpoint is provisioned and ready to use. You can now start to publish and subscribe to topics.

Endpoint

rest-2.amazonaws.com

**Logs** ENABLED

You can enable AWS IoT to log helpful information to CloudWatch Logs. As messages from your devices pass through the message broker and the rules engine, AWS IoT logs process events which can be helpful in troubleshooting.

**Role**

my-iot-role

**Level of verbosity**

Debug

[Edit](#)

**Event-based messages** ENABLED

AWS IoT can send event-based messages to pre-determined MQTT topics when specific service events occur.

Event	Publish to MQTT	MQTT topic	Subscribe all
Job: completed, canceled	● Enabled	<a href="#">Copy topic</a>	<a href="#">Subscribe</a>
Job execution: success, failed, rejected, canceled, removed	● Enabled	<a href="#">Copy topic</a>	<a href="#">Subscribe</a>

- **Client ID:** any unique ID.
- **Certificate For Thing:** thing certificate or custom certificate downloaded when the created thing is imported.
- **Private Key:** private key or custom private key downloaded when the created thing is imported.
- **rootCA:** CA certificate imported for server authentication. You can download the CA certificate from . It is recommended to use **Amazon Root CA 1** or **Starfield Starfield root CA certificate**. Currently, the Amazon Root CA 3 certificate is not supported.
- Use the default values for other parameters.

The screenshot shows the AWS IoT Developer Guide page for 'CA certificates for server authentication'. The left sidebar contains a navigation menu with categories like 'What is AWS IoT?', 'Getting started with AWS IoT Core', 'AWS IoT Tutorials', 'Managing devices with AWS IoT', 'Tagging your AWS IoT resources', and 'Security'. Under 'Security', 'Authentication' is expanded, showing 'Server authentication' as the active section. The main content area explains that certificates are signed by one of the following root CA certificates:

- VeriSign Endpoints (legacy)**
  - RSA 2048 bit key: [VeriSign Class 3 Public Primary G5 root CA certificate](#)
- Amazon Trust Services Endpoints (preferred)**
  - RSA 2048 bit key: [Amazon Root CA 1](#)
  - RSA 4096 bit key: Amazon Root CA 2. Reserved for future use.
  - ECC 256 bit key: [Amazon Root CA 3](#)
  - ECC 384 bit key: Amazon Root CA 4. Reserved for future use.

A note states: 'You might need to right click these links and select **Save link as...** to save these certificates as files.' Below the list, it mentions that all certificates are cross-signed by the [Starfield Root CA Certificate](#) and that new AWS IoT Core regions serve only ATS certificates.

## 2.2 Publishing Messages to the AWS IoT

- Step 1: Configure the message to be published.

Choose **Cloud > Message Management** and add the message to be published. The following figure shows the configuration:

The screenshot shows the 'Publish' configuration page in the AWS IoT console. The breadcrumb trail is 'Overview / Edge Computing / Device Supervisor / Cloud'. The configuration fields are as follows:

- Name:** data\_upload
- Topic:** awsiot/test
- Qos(MQTT):** 1
- Group Type:** Collect (selected), Alarm
- Group:** default
- Main Function:** vars\_upload\_test (with a note: 'Should be the same with the name of the entry function in the script')
- Script:** A code editor containing the following Python script:
 

```
1 import logging
2 from datetime import datetime
3
4
5 def vars_upload_test(data_collect, wizard_api):
6     value_list = []
7     for device, val_dict in data_collect['values'].items():
8         value_dict = {
9             "Device": device,
10            "timestamp": data_collect["timestamp"],
11            "Data": {}
12        }
13        for id, val in val_dict.items():
14            value_dict["Data"][id] = val["raw_data"]
15        value_list.append(value_dict)
16        logging.info(value_list)
17    wizard_api.aws_iot_publish("awsiot/test", value_list, 1)
```

At the bottom, there are 'Submit' and 'Reset' buttons.

The script is as follows:

```

import logging
from datetime import datetime
"""
Logs are generally generated in the gateway in the following ways:
1. import logging: uses logging.info(XXX) to generate logs. Display of logs
↳ generated in this way is not controlled by the log level parameter on the global
↳ parameter page.
2. from common.Logger import logger: uses logger.info(XXX) to generate logs.
↳ Display of logs generated in this way is controlled by the log level parameter on
↳ the global parameter page.
"""

def vars_upload_test(data_collect, wizard_api): # Define the main publishing
↳ function.
    value_list = [] # Define the data list.
    for device, val_dict in data_collect['values'].items(): # Traverse the values
↳ dictionary. The dictionary contains the device name and the variables of the
↳ device.
        value_dict = { # Customize the data dictionary.
            "Device": device,
            "timestamp": data_collect["timestamp"],
            "Data": {}
        }
        for id, val in val_dict.items(): # Traverse variables and assign values for
↳ the Data dictionary.
            value_dict["Data"][id] = val["raw_data"]
            value_list.append(value_dict) # Add data in value_dict to value_list in
↳ sequence.
        logging.info(value_list) # Print data in value_list in app logs in the
↳ following format: [{'Device': 'S7-1200', 'timestamp': 1589538347.5604711, 'Data':
↳ {'Test1': False, 'Test2': 12}}].
    return value_list # Send value_list to the app, which then uploads it to the
↳ MQTT server by collection time. If it fails to be sent, cache the data and upload
↳ it to the MQTT server by collection time after the connection recovers.

```

The message publishing parameters are described as follows:

- Name: custom publication name.
- Topic: publication topic, which must be consistent with the topic that the MQTT server subscribes to.
- Qos(MQTT): publication QoS, which is recommended to be consistent with that of the MQTT

server.

- \* 0: The message is sent only once, without retry.
- \* 1: The message is sent at least once to ensure that it reaches the MQTT server.
- **Group Type:** when publishing variable data, select **Collection**. Then, only **Collection Group** is available in **Group**. When publishing alarm data, select **Alarm**. Then, only **Alarm Group** is available in **Group**.
- **Group:** after a group is selected, all variables in this group are uploaded to the MQTT server according to the publication configuration. If you select multiple groups, the script logic in the publication is executed for the variables in each group at the collection interval of the groups. The group must include variables. Otherwise, the script logic in the publication is not executed.
- **Main Function:** name of the main function (entry function), which must be consistent with that in the script.
- **Script:** uses Python code to customize the packaging and processing logic. The main function parameters are as follows:
  - \* **Parameter 1:** same as **Parameter 1** in the main function of [Standard MQTT-Publishing](#).
  - \* **Parameter 2:** AWS IoT API of the Device Supervisor. For details, see *Device Supervisor AWS IoT API Description*.
- Step 2: Subscribe to messages in the AWS IoT.

Choose **AWS IoT > Test** and enter the IG902 publication topic in the **Subscription topic** text box. As an example, the topic is `awsiot/test`.

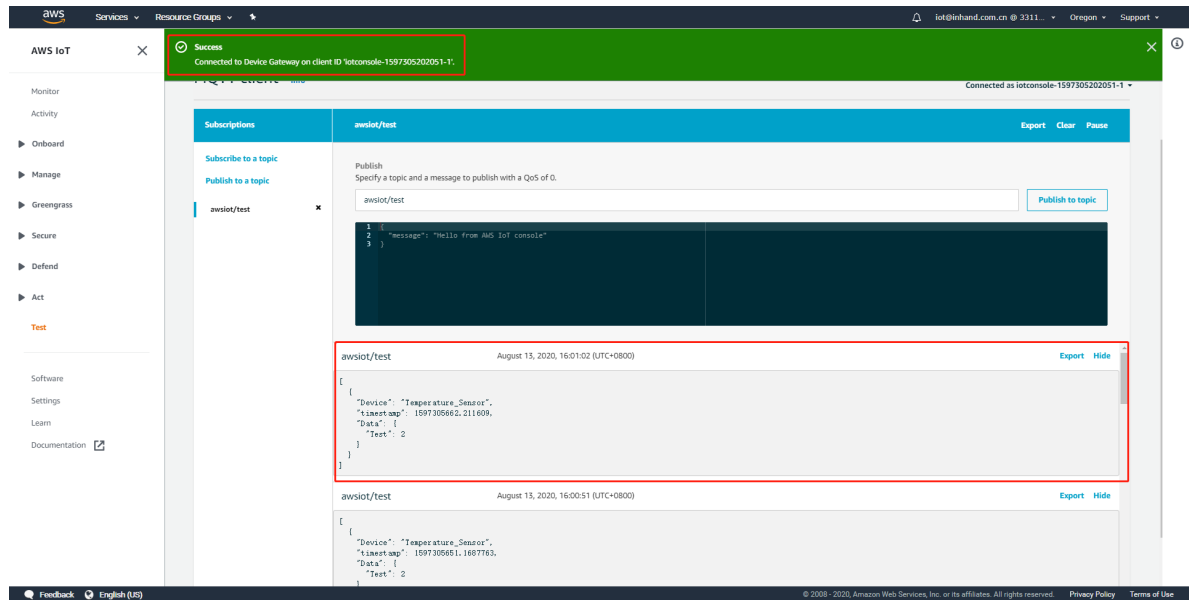
The screenshot shows the AWS IoT console interface. At the top, a green banner indicates a successful connection to the Device Gateway. The left sidebar shows navigation options, with 'Test' highlighted. The main panel displays the 'MQTT client' interface. Under the 'Subscriptions' tab, the 'Subscription topic' is set to 'awsiot/test'. The 'Subscribe to topic' button is highlighted. Below this, the 'Quality of Service' is set to 0, and the 'MQTT payload display' is set to 'Auto-format JSON payloads'. At the bottom, the 'Message Management' section shows a table with one entry: 'data\_upload' with topic 'awsiot/test'.

Name	Type	Topic	Qos(MQTT)	Group	Main Function	Operation
data_upload		awsiot/test	1	default	vars_upload_test	

- Step 3: View the messages that the AWS IoT receives.



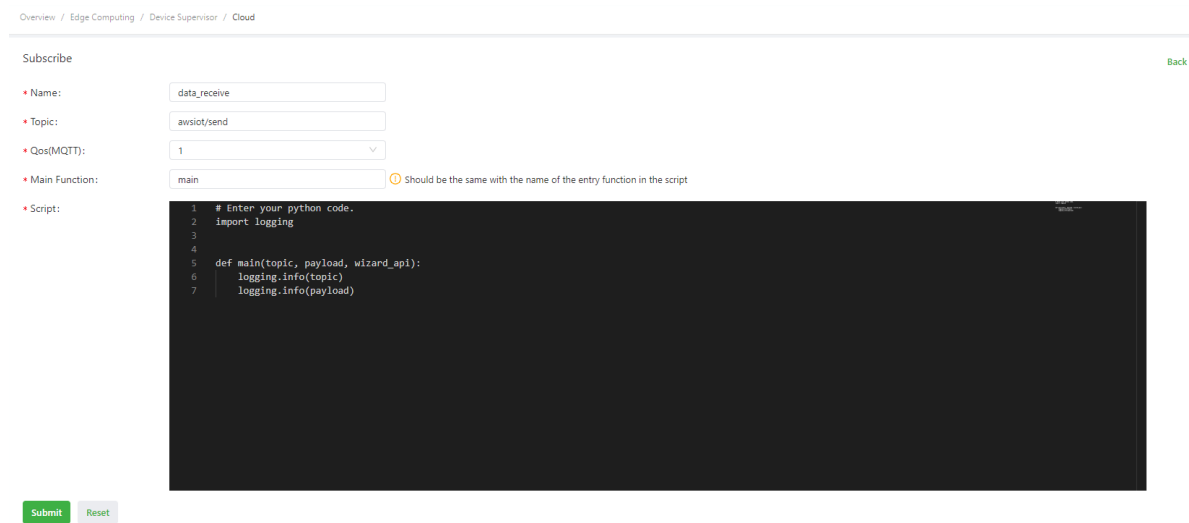
After subscribing to the topic, you can view the message content under the topic.



## 2.3 Subscribing to AWS IoT Messages

- Step 1: Configure the message for subscription.

Choose **Cloud > Message Management** and add the message for subscription. The following figure shows the configuration:



The message subscription parameters are described as follows:

- **Name:** custom subscription name.
- **Topic:** subscription topic, which must be consistent with the data topic published by the MQTT server.

- **Qos(MQTT)**: subscription QoS. The default value is 0.
- **Main Function**: name of the main function (entry function), which must be consistent with that in the script.
- **Script**: uses Python code to customize the packaging and processing logic. The main function parameters of custom topic subscription are as follows:
  - \* **Parameter 1**: received topic. The data type is **string**.
  - \* **Parameter 2**: received data. The data type is **string**.
  - \* **Parameter 3**: AWS IoT API of the Device Supervisor. For details, see *Device Supervisor AWS IoT API Description*.
- Step 2: Publish messages in the AWS IoT.

Choose **AWS IoT > Test** and enter the IG902 subscription topic in the **Publish to topic** text box. As an example, the topic is **awsiot/send**.

**Message Management**

Publish

Name	Type	Topic	Qos(MQTT)	Group	Main Function	Operation
data_upload		awsiot/test	1	default	vars_upload_test	

Subscribe

Name	Type	Topic	Qos(MQTT)	Main Function	Operation
data_receive		awsiot/send	1	main	

- Step 3: View the messages that the AWS IoT publishes.

After the AWS IoT publishes messages, you can view the received messages in the run logs of the app.

```
[2020-08-13 16:03:51.387] [INFO] [AWSIoT.py 200]: [AWSIoT]: on_cloud_subscribe. topic: awsiot/send, payload: b'{"message": "Hello from AWS IoT console"}', qos: 0
[2020-08-13 16:03:51.399] [INFO] [<string> 6]: awsiot/send
[2020-08-13 16:03:51.401] [INFO] [<string> 7]: b'{"message": "Hello from AWS IoT console"}'
[2020-08-13 16:03:51.402] [INFO] [AWSIoT.py 142]: [AWSIoT]: receive message, topic: awsiot/send, payload: b'{"message": "Hello from AWS IoT console"}'
```

## 1.2.4 Appendix

## Device Supervisor AWS IoT API Description

For details about the basic configuration of `wizard_api`, see [Device Supervisor API Description](#). If the cloud service type is AWS IoT, `wizard_api` additionally provides the following method:

- `awsiot_publish(topic, payload, qos)`
  - Method Description: data submitting method.
  - Parameter
    - \* Parameter 1: MQTT topic. The data type is `string`. This topic is used to send the data to the MQTT server.
    - \* Parameter 2: data to be sent.
    - \* Parameter3: QoS level. The options are 0 and 1.
  - Usage example:

Overview / Edge Computing / Device Supervisor / Cloud

Back

Publish

\* Name:

\* Topic:

\* Qos(MQTT):

Group Type: ☒ Collect ☐ Alarm

\* Group:

\* Main Function:  Should be the same with the name of the entry function in the script

\* Script:

```

1 import logging
2 from datetime import datetime
3
4 def vars_upload_test(data_collect, wizard_api):
5     value_list = []
6     for device, val_dict in data_collect['values'].items():
7         value_dict = {
8             "Device": device,
9             "timestamp": data_collect["timestamp"],
10            "Data": {}
11        }
12        for id, val in val_dict.items():
13            value_dict["Data"][id] = val["raw_data"]
14        value_list.append(value_dict)
15        logging.info(value_list)
16        wizard_api.awsiot_publish("awsiot/test", value_list, 1)

```

Submit

Reset

```

import logging
from datetime import datetime
"""

```

*Logs are generally generated in the gateway in the following ways:*

1. *import logging: uses `logging.info(XXX)` to generate logs. Display of logs*  
*↳ generated in this way is not controlled by the log level parameter on the*  
*↳ global parameter page.*
2. *from common.Logger import logger: uses `logger.info(XXX)` to generate logs.*  
*↳ Display of logs generated in this way is controlled by the log level*  
*↳ parameter on the global parameter page.*

```

"""

```

(continues on next page)

(continued from previous page)

```

def vars_upload_test(data_collect, wizard_api): # Define the main publishing
↪function.
    value_list = [] # Define the data list.
    for device, val_dict in data_collect['values'].items(): # Traverse the
↪values dictionary. The dictionary contains the device name and the variables
↪of the device.
        value_dict = { # Customize the data dictionary.
            "Device": device,
            "timestamp": data_collect["timestamp"],
            "Data": {}
        }
        for id, val in val_dict.items(): # Traverse variables and assign values
↪for the Data dictionary.
            value_dict["Data"][id] = val["raw_data"]
        value_list.append(value_dict) # Add data in value_dict to value_list in
↪sequence.
        logging.info(value_list) # Print data in value_list in app logs in the
↪following format: [{'Device': 'S7-1200', 'timestamp': 1589538347.5604711,
↪'Data': {'Test1': False, 'Test2': 12}}].
        wizard_api.aws_iot_publish("aws_iot/test", value_list, 1) # Send value_list
↪to the app, which then uploads it to the MQTT server by collection time. If
↪it fails to be sent, cache the data and upload it to the MQTT server by
↪collection time after the connection recovers.

```

## 1.3 Azure IoT User Manual

The Azure IoT Hub (Azure IoT for short) is hosted in the cloud as the central message center for bidirectional communication between IoT applications and devices managed by them. You can establish reliable and secure communication between millions of IoT devices and backend of cloud-hosting solutions and generate IoT solutions through the Azure IoT. This allows users to connect any device to the IoT Hub.

The edge computing gateway InGateway902 (IG902 for short) provides the Device Supervisor app (Device Supervisor for short) to help users connect their devices to the Azure IoT. This document uses IG902 as an example to describe how to submit service data and deliver configuration data between the Device Supervisor and the Azure IoT.

- *Prerequisites*
- *1. Environment Preparation*
  - *1.1 Configuring the Azure IoT*

- \* *1.1.1 Adding the IoT Hub*
    - \* *1.1.2 Adding the IoT Device*
  - *1.2 Configuring the Edge Computing Gateway*
    - \* *1.2.1 Basic Configuring*
    - \* *1.2.2 Data Collecting Configuration*
- *2. Message Publishing and Subscription*
  - *2.1 Connecting to the Azure IoT*
  - *2.2 Publishing Messages to the Azure IoT*
  - *2.3 Subscribing to Azure IoT Messages*
- *Appendix*
  - *Example of Publishing Messages to the Azure IoT*
  - *Example of Subscribing to Azure IoT Messages*
  - *Device Supervisor Azure IoT API Description*
- *FAQ*
  - *Q1: The Azure IoT Connection Frequently Fails Shortly After It Is Established*

### 1.3.1 Prerequisites

- Azure cloud platform account
- Edge computing gateway IG501/IG902
  - Firmware version
    - \* IG902: IG9-V2.0.0.r12754 or later
    - \* IG501: IG5-V2.0.0.r12884 or later
  - SDK version
    - \* IG902: py3sdk-V1.4.0\_Edge-IG9 or later
    - \* IG501: py3sdk-V1.4.0\_Edge-IG5 or later
  - App version: device\_supervisor-V1.2.5 or later

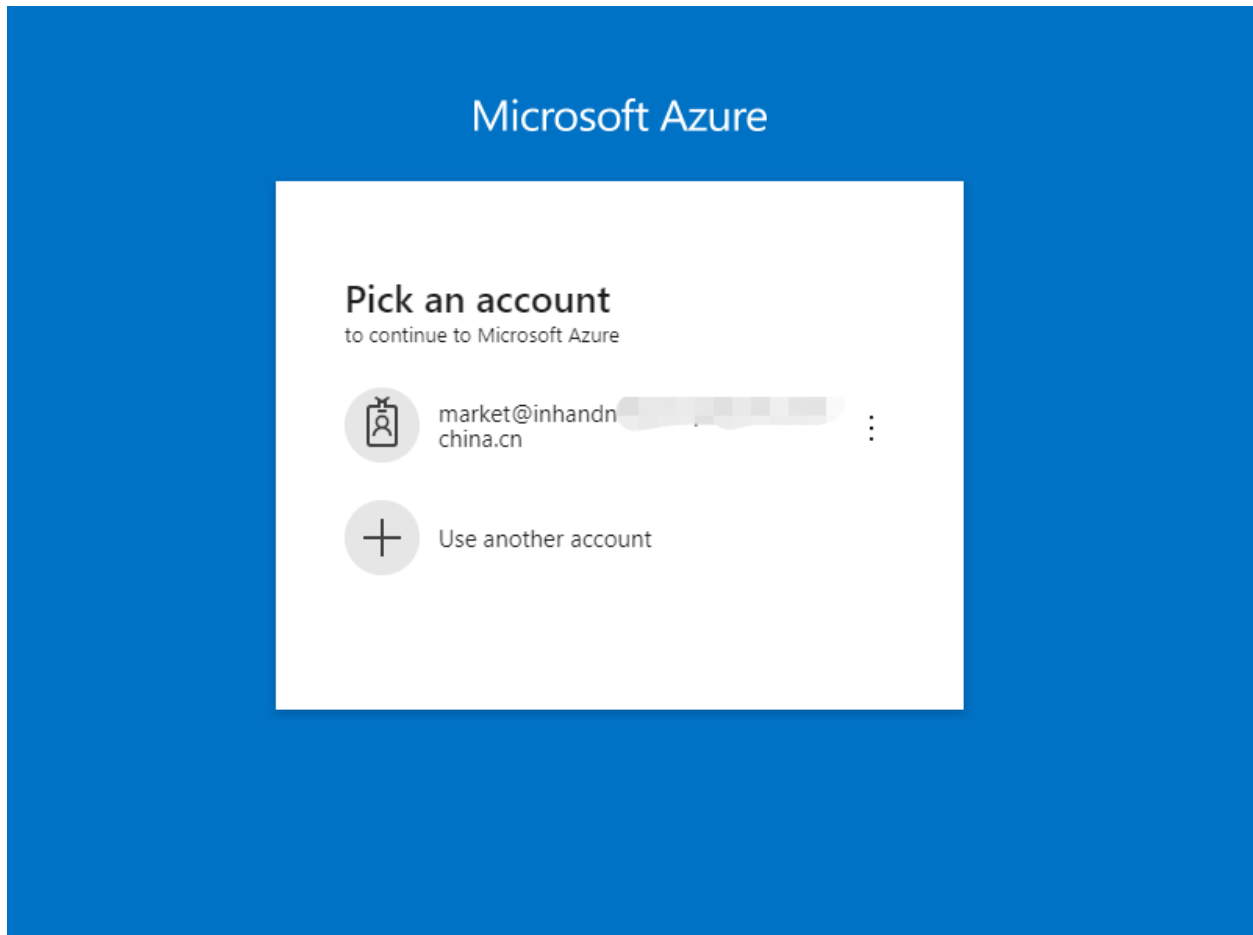
### 1.3.2 1. Environment Preparation

- *1.1 Configuring the Azure IoT*
- *1.2 Configuring the Edge Computing Gateway*

## 1.1 Configuring the Azure IoT

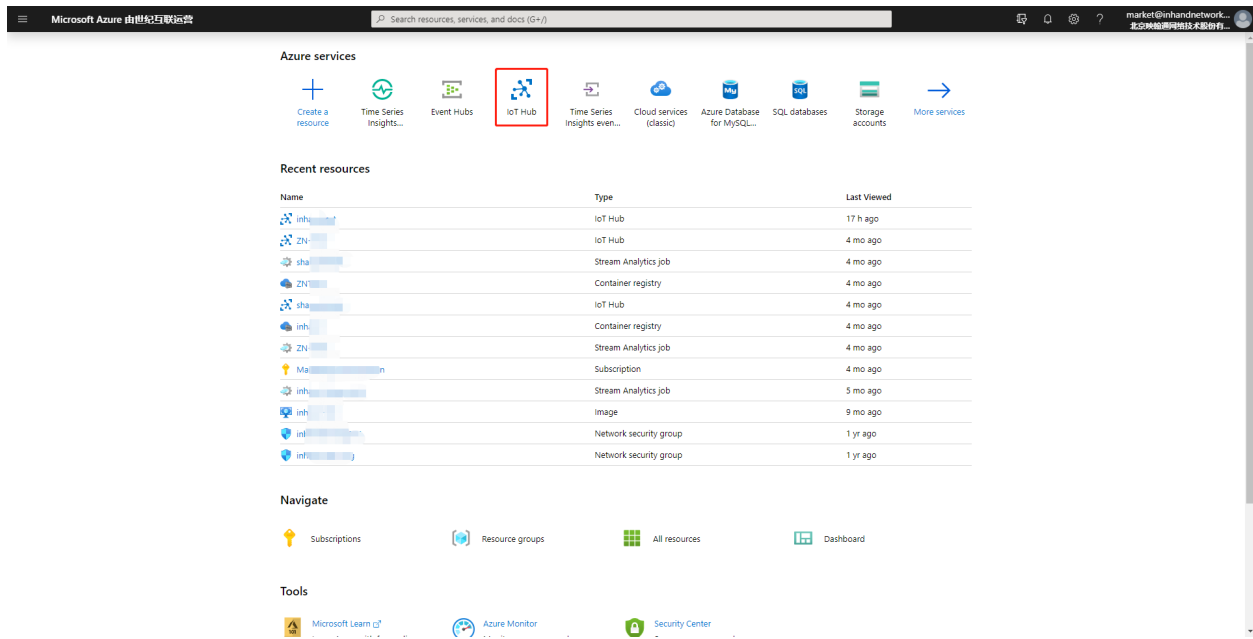
- 1.1.1 Adding the IoT Hub
- 1.1.2 Adding the IoT Device

If you have configured the IoT Hub and IoT device in the Azure IoT, go to *1.2 Configuring the Edge Computing Gateway*. Otherwise, perform the following steps to configure the Azure IoT. Visit <https://portal.azure.cn/> to log in to Azure.

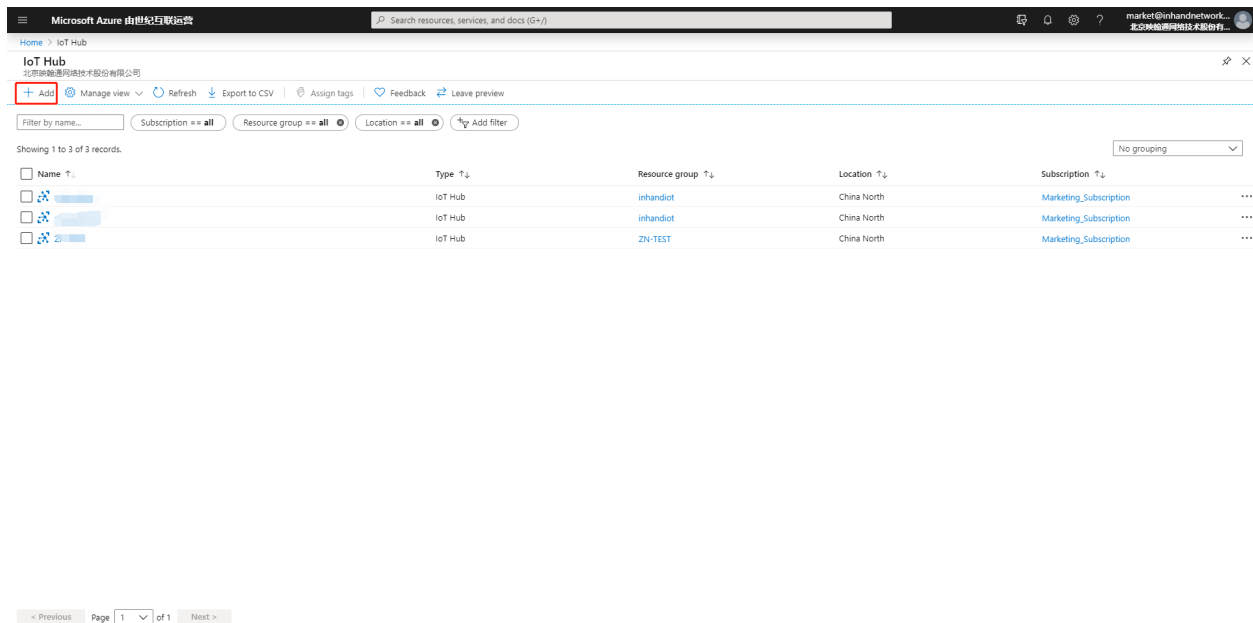


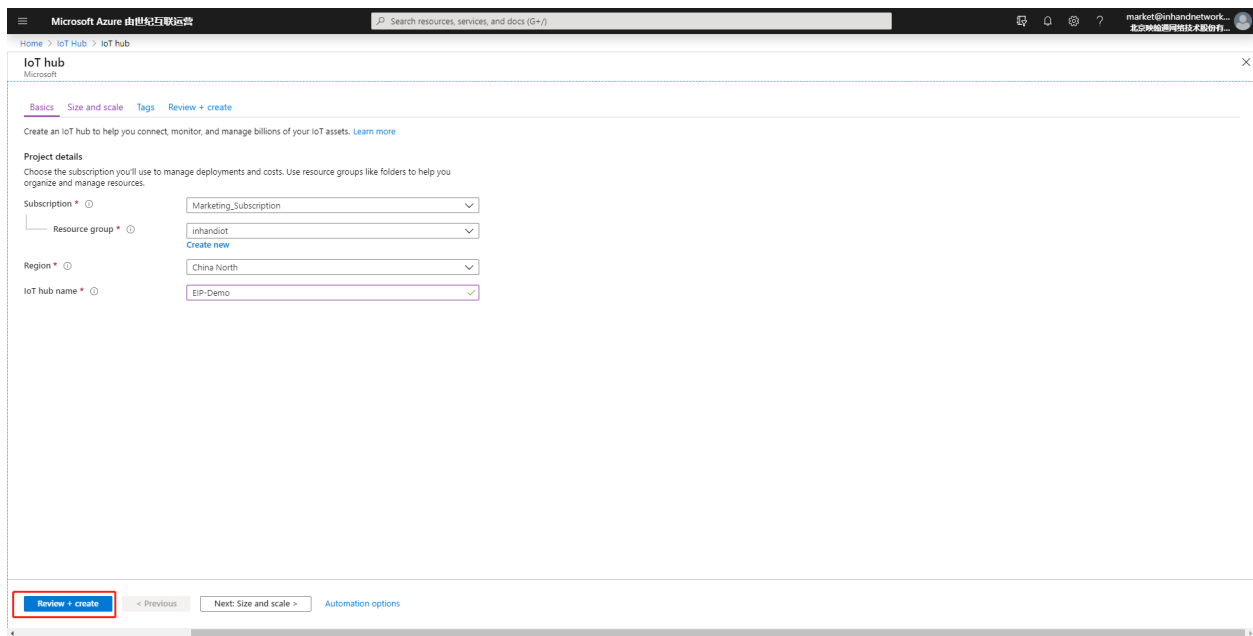
### 1.1.1 Adding the IoT Hub

Choose **IoT Hub** after login, as shown in the following figure:

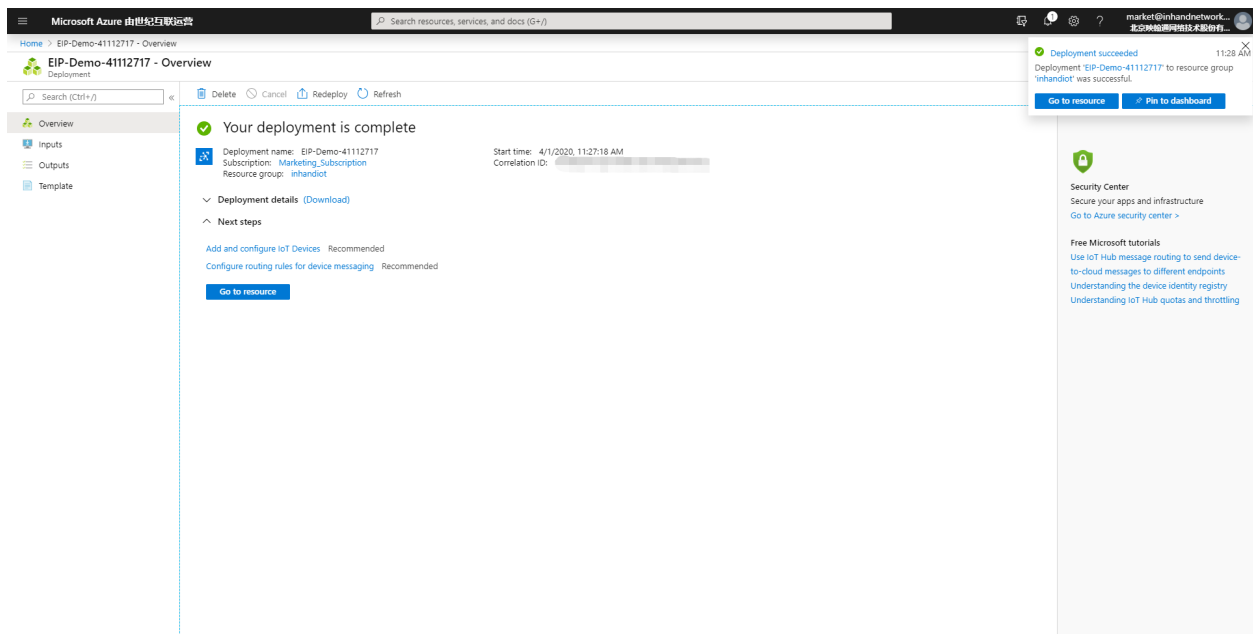


Click **Add** to create an IoT Hub.





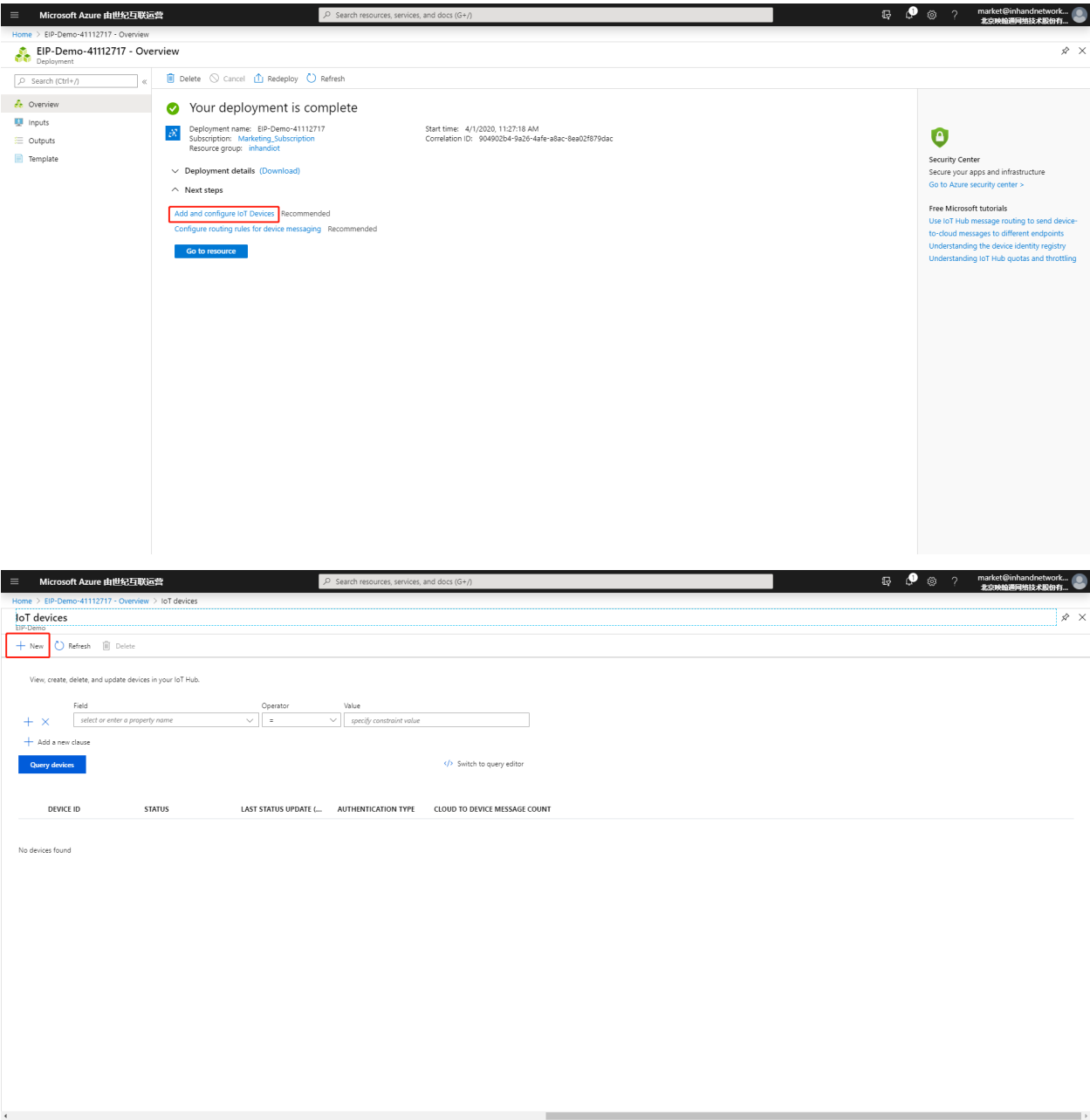
The following figure is displayed after the IoT Hub is created:



### 1.1.2 Adding the IoT Device

Create an IoT device in the IoT Hub.





Microsoft Azure 由世纪互联运营

Home > EIP-Demo-41112717 - Overview > IoT devices > Create a device

Value  
specify constraint value

Switch to query editor

AUTHENTICATION TYPE CLOUD TO DEVICE MESSAGE COUNT

Create a device

Find Certified for Azure IoT devices in the Device Catalog

Device ID \*  
EIP-Demo-Test

Authentication type  
Symmetric key X.509 Self-Signed X.509 CA Signed

Primary key  
Enter your primary key

Secondary key  
Enter your secondary key

Auto-generate keys  
☒

Connect this device to an IoT hub  
☒ Enable Disable

Parent device  
No parent device  
Set a parent device

Save

The following figure is displayed after the IoT device is created:

Microsoft Azure 由世纪互联运营

Home > EIP-Demo-41112717 - Overview > IoT devices

IoT devices  
EIP-Demo

+ New Refresh Delete

View, create, delete, and update devices in your IoT Hub.

+ × Field Operator Value  
select or enter a property name = specify constraint value

+ Add a new clause

Query devices. Switch to query editor

DEVICE ID	STATUS	LAST STATUS UPDATE	AUTHENTICATION TYPE	CLOUD TO DEVICE MESSAGE COUNT
EIP-Demo-Test	Enabled	--	Sas	0

## 1.2 Configuring the Edge Computing Gateway

- 1.2.1 Basic Configuring
- 1.2.2 Data Collecting Configuration

### 1.2.1 Basic Configuring

- For details about IG902 connection configuration and software version update, see [IG902 Quick Guide](#).
- For details about IG501 connection configuration and software version update, see [IG501 Quick Guide](#).

### 1.2.2 Data Collecting Configuration

For details about the basic data collection configuration for the Device Supervisor, see [Device Supervisor App User Manual](#). The following figure shows the data collection configuration in this document:

Overview / Edge Computing / Device Supervisor / Device List

Device List

Temperature\_Sensor

ModbusTCP

IP: 10.5.16.82

+

✎

Operation :

+

↓

↑

✎

Total 1 item

<

1

>

Variables Table(Temperature\_Sensor)

Input Variable Name

🔍

Operation :

↓

↑

<input type="checkbox"/>	Name	Group	Data Type	Address	Value	Description	Time	Operation
<input type="checkbox"/>	• Temperature	default	WORD	40001	221 °C	✎	2020-08-13 16:13:02	✎ ✎
<input type="checkbox"/>	• Humidity	default	WORD	40002	521		2020-08-13 16:13:02	✎ ✎

+ Add to Group

✎ Delete

Total 2 items

<

1

>

50 / page ▾

## 1.3.3 2. Message Publishing and Subscription

- *2.1 Connecting to the Azure IoT*
- *2.2 Publishing Messages to the Azure IoT*
- *2.3 Subscribing to Azure IoT Messages*

### 2.1 Connecting to the Azure IoT

Choose **Edge Computing > Device Supervisor > Cloud** on IG902, select **Enable Cloud Service**, and select **Azure IoT** from the Type drop-down list. The following is a configuration example:

Overview / Edge Computing / Device Supervisor / Cloud

## Status

Cloud Status: Connection Successful

Connection time: 0 Day 00:13:31

### Enable Cloud Service:



\* Type:

Azure IoT

\* Connection String:

.....



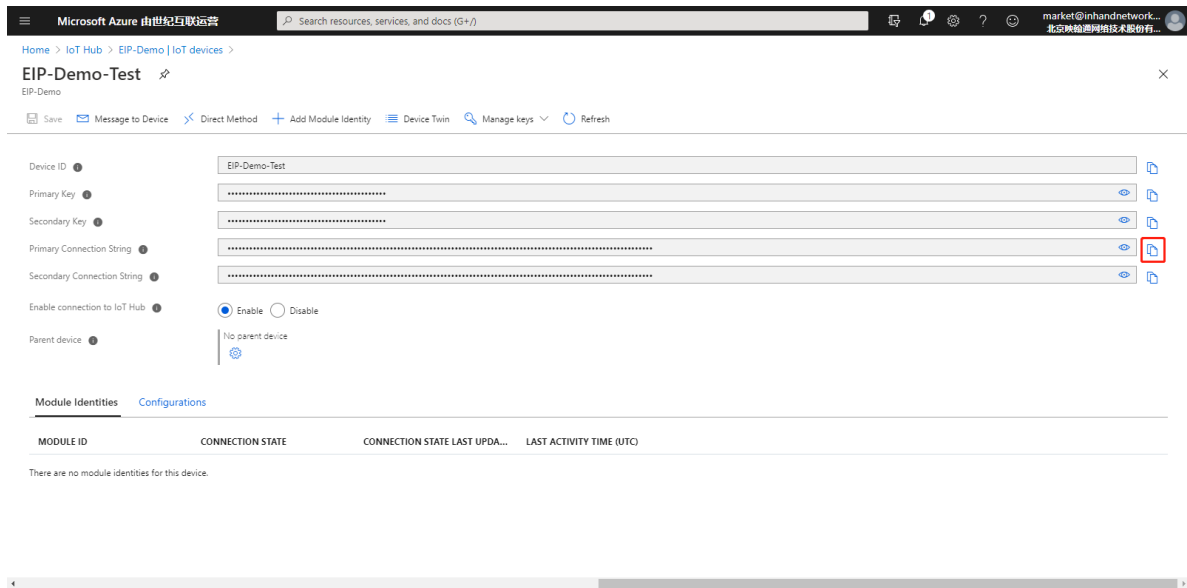
Submit

Reset

The parameters are described as follows:

- **Type:** select **Azure IoT** for an Azure IoT connection.
- **Connection String:** main connection string of the Azure IoT device. You can select the device from the IoT Hub in the Azure IoT and copy the main connection string to here.

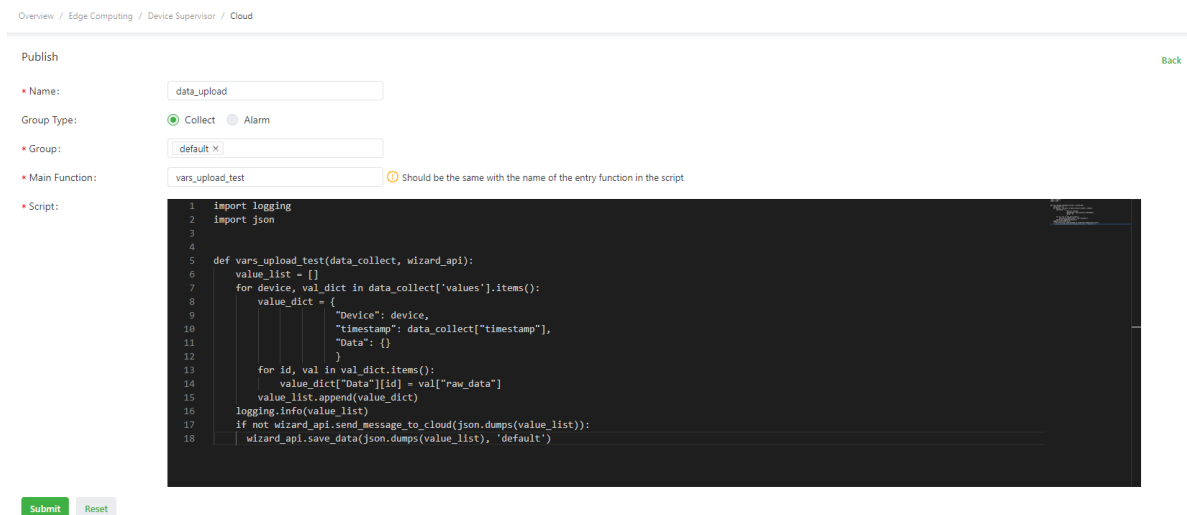
DEVICE ID	STATUS	LAST STATUS UPDATE	AUTHENTICATION TYPE	CLOUD TO DEVICE MESSAGE COUNT
EIP-Demo-Test	Enabled	--	Sas	0



## 2.2 Publishing Messages to the Azure IoT

- Step 1: Configure the message to be published.

Choose **Cloud > Message Management** and add the message to be published. The following figure shows the configuration:



```
import logging
```

```
"""
```

*Logs are generally generated in the gateway in the following ways:*

*1. import logging: uses logging.info(XXX) to generate logs. Display of logs*

*↳ generated in this way is not controlled by the log level parameter on the global*

*↳ parameter page.*

(continues on next page)

(continued from previous page)

```

2. from common.Logger import logger: uses logger.info(XXX) to generate logs.
↪Display of logs generated in this way is controlled by the log level parameter on
↪the global parameter page.
"""

def vars_upload_test(data_collect, wizard_api): # Define the main publishing
↪function.
    value_list = [] # Define the data list.
    for device, val_dict in data_collect['values'].items(): # Traverse the values
↪dictionary. The dictionary contains the device name and the variables of the
↪device.
        value_dict = { # Customize the data dictionary.
            "Device": device,
            "timestamp": data_collect["timestamp"],
            "Data": {}
        }

        for id, val in val_dict.items(): # Traverse variables and assign values for
↪the Data dictionary.
            value_dict["Data"][id] = val["raw_data"]
            value_list.append(value_dict) # Add data in value_dict to value_list in
↪sequence.

        logging.info(value_list) # Print data in value_list in app logs in the
↪following format: [{'Device': 'S7-1200', 'timestamp': 1589538347.5604711, 'Data':
↪{'Test1': False, 'Test2': 12}}].

    return value_list # Send value_list to the app, which then uploads it to the
↪MQTT server by collection time. If it fails to be sent, cache the data and upload
↪it to the MQTT server by collection time after the connection recovers.

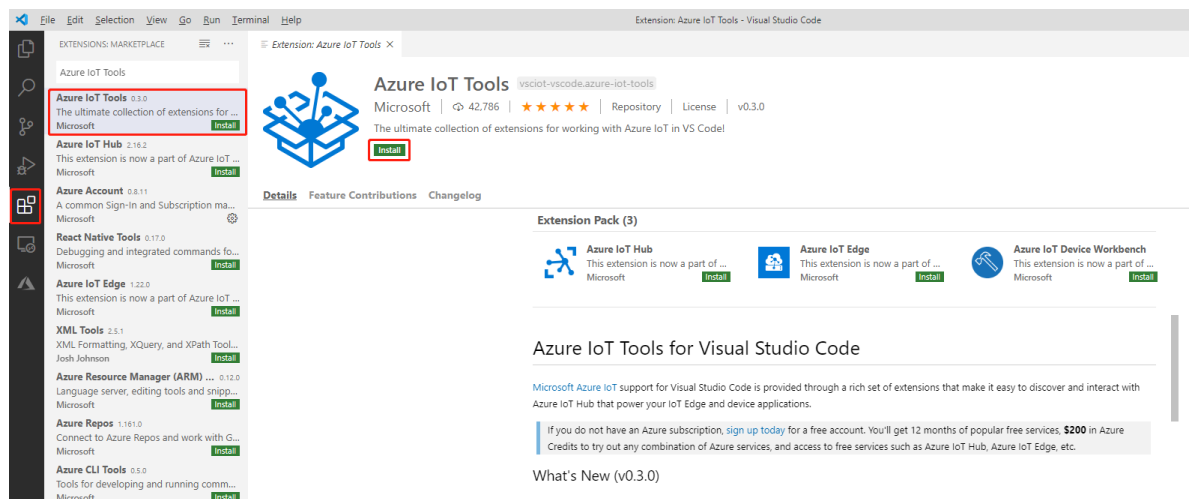
```

The message publishing parameters are described as follows:

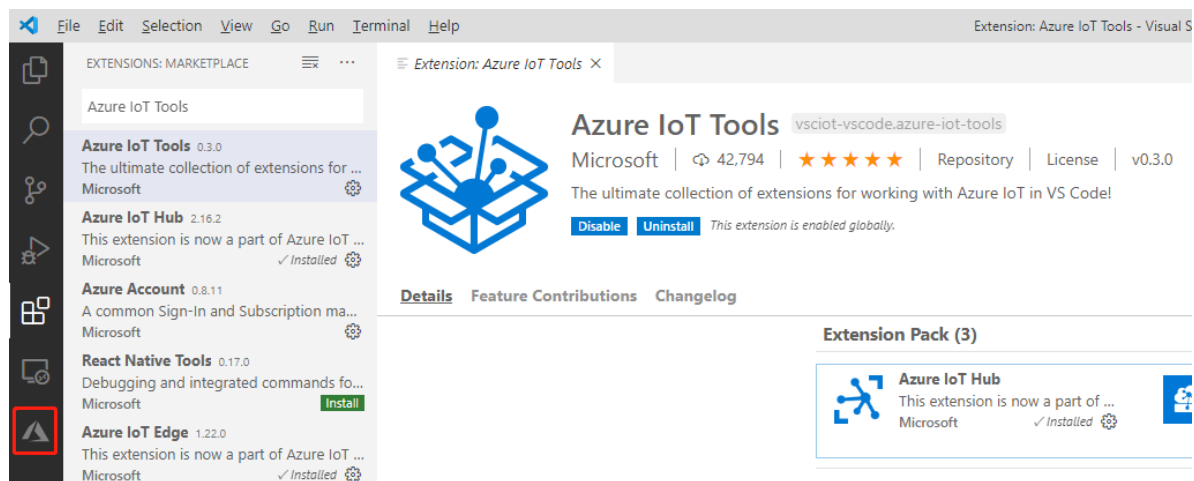
- **Name:** custom publication name.
- **Group Type:** when publishing variable data, select **Collection**. Then, only **Collection Group** is available in **Group**. When publishing alarm data, select **Alarm**. Then, only **Alarm Group** is available in **Group**.
- **Group:** after a group is selected, all variables in this group are uploaded to the MQTT server according to the publication configuration. If you select multiple groups, the script logic in the publication is executed for the variables in each group at the collection interval of the groups. The group must include variables. Otherwise, the script logic in the publication is not executed.
- **Main Function:** name of the main function (entry function), which must be consistent with that in the script.

- **Script:** uses Python code to customize the packaging and processing logic. The main function parameters are as follows:
  - \* **Parameter 1:** same as **Parameter 1** in the main function of **Standard MQTT-Publishing**.
  - \* **Parameter 2:** Azure IoT API of the Device Supervisor. For details, see *Device Supervisor Azure IoT API Description*.
- **Step 2:** Use the Azure IoT Tools plug-in of the Visual Studio Code (VS Code for short) to establish the connection to the IoT Hub.

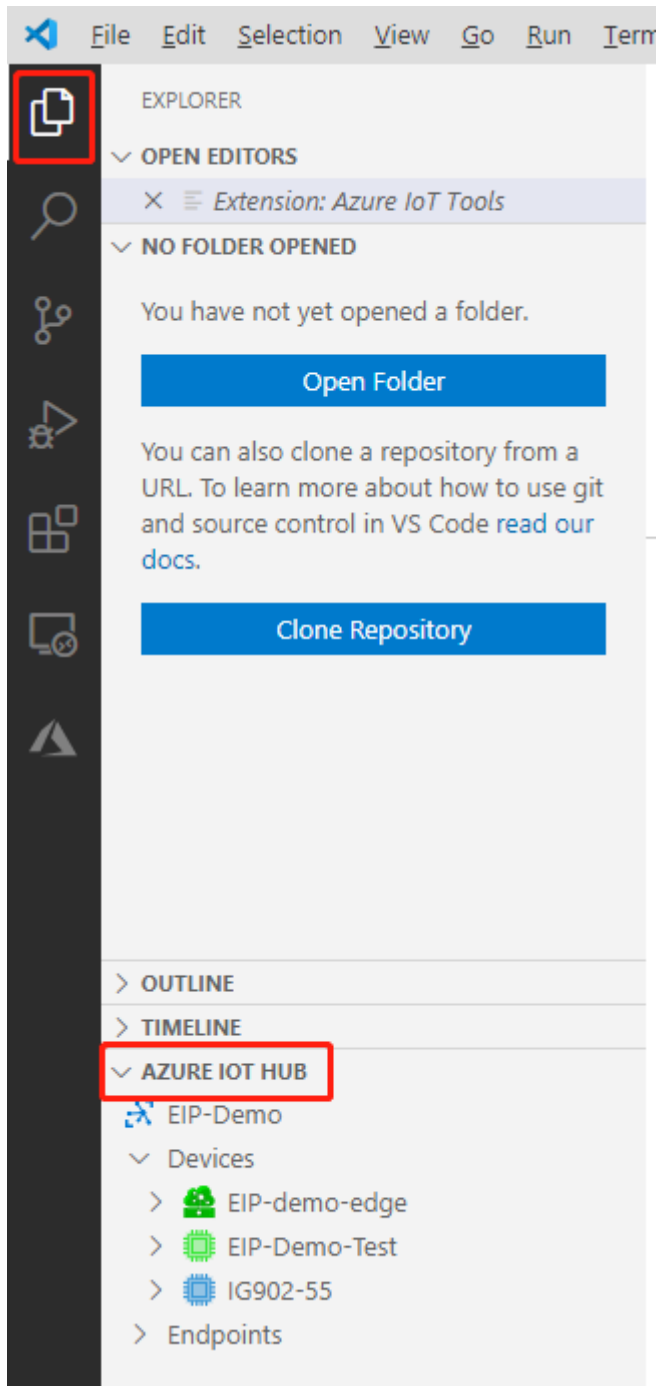
After submitting the published message, use the Azure IoT Tools plug-in of the VS Code to view messages sent to the Azure IoT. You can download the VS Code from <https://code.visualstudio.com/Download>. Install and start the VS Code, choose **Extensions**, search for **Azure IoT Tools**, and then install the **Azure IoT Tools** plug-in.



The **Azure** module appears on the left if the installation is successful.

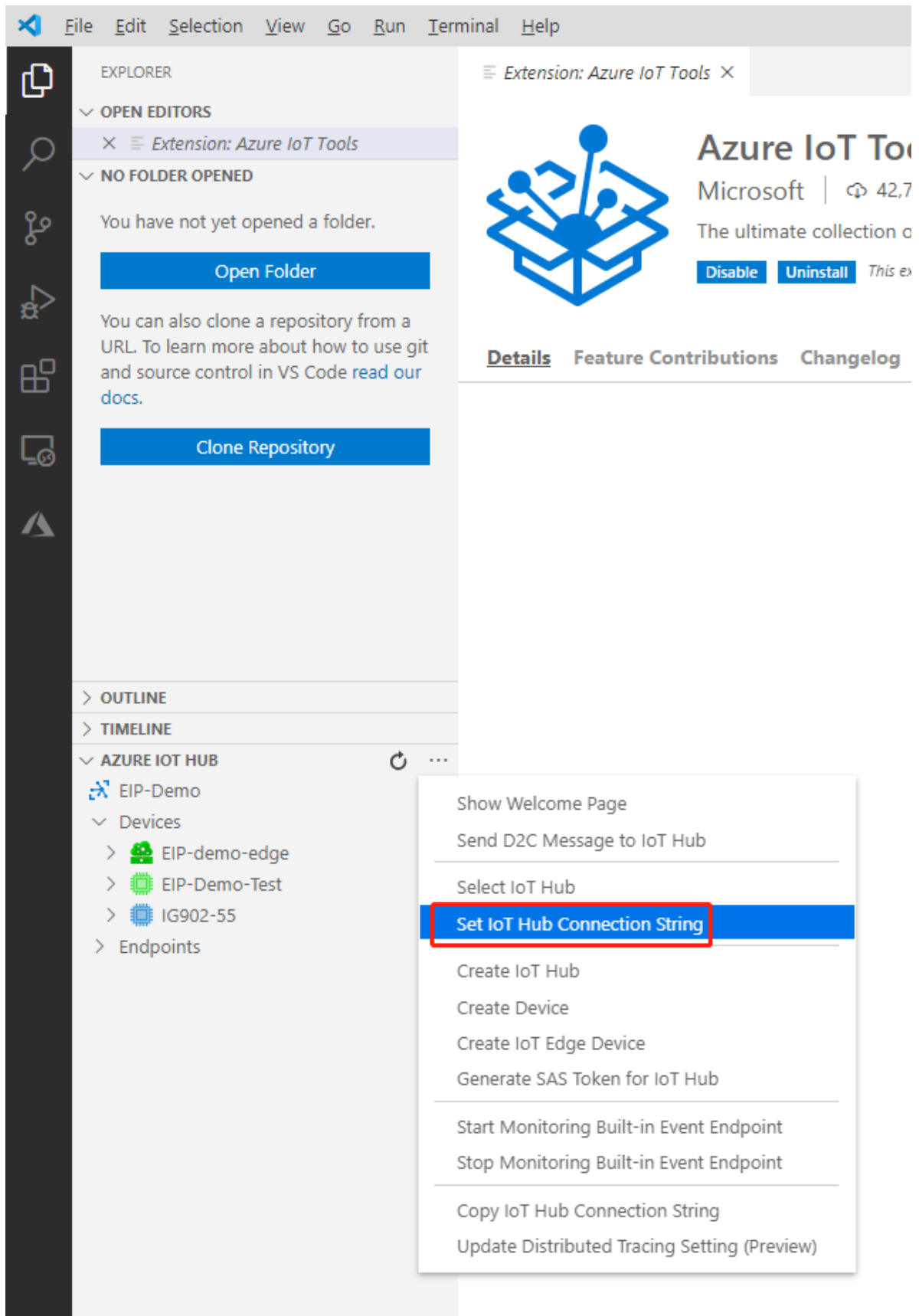


Choose **Explorer > AZURE IOT HUB** to expand **AZURE IOT HUB**.



Click ... and choose **Set IoT Hub Connection String**.



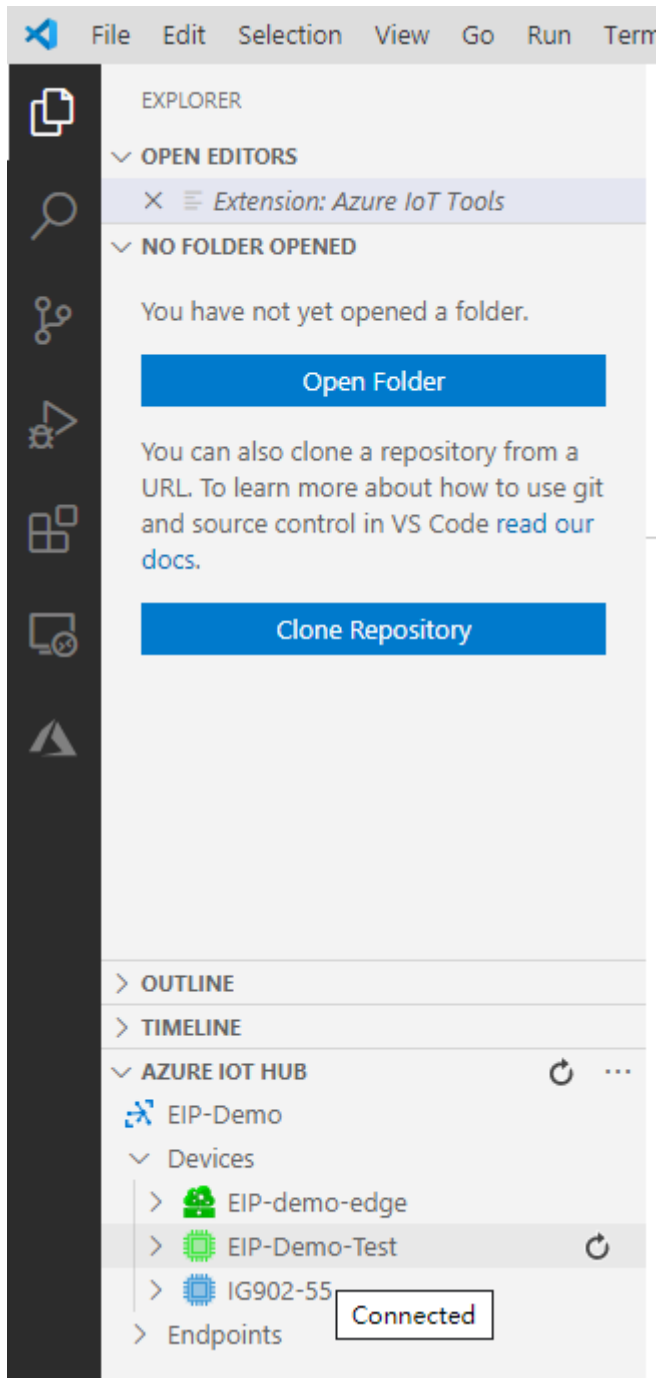


Enter the IoT Hub connection string in the text box. You can go to the specified IoT Hub and choose **Shared access policies** > **iothubowner** to obtain the IoT Hub connection string.

The top screenshot shows the Azure portal interface for an IoT Hub named "EIP-Demo". In the left-hand navigation pane, "Shared access policies" is selected. The main pane displays a table of policies, with "iothubowner" selected. On the right, the details for the "iothubowner" policy are shown, including its permissions and shared access keys. The "Primary key" connection string is highlighted with a red box.

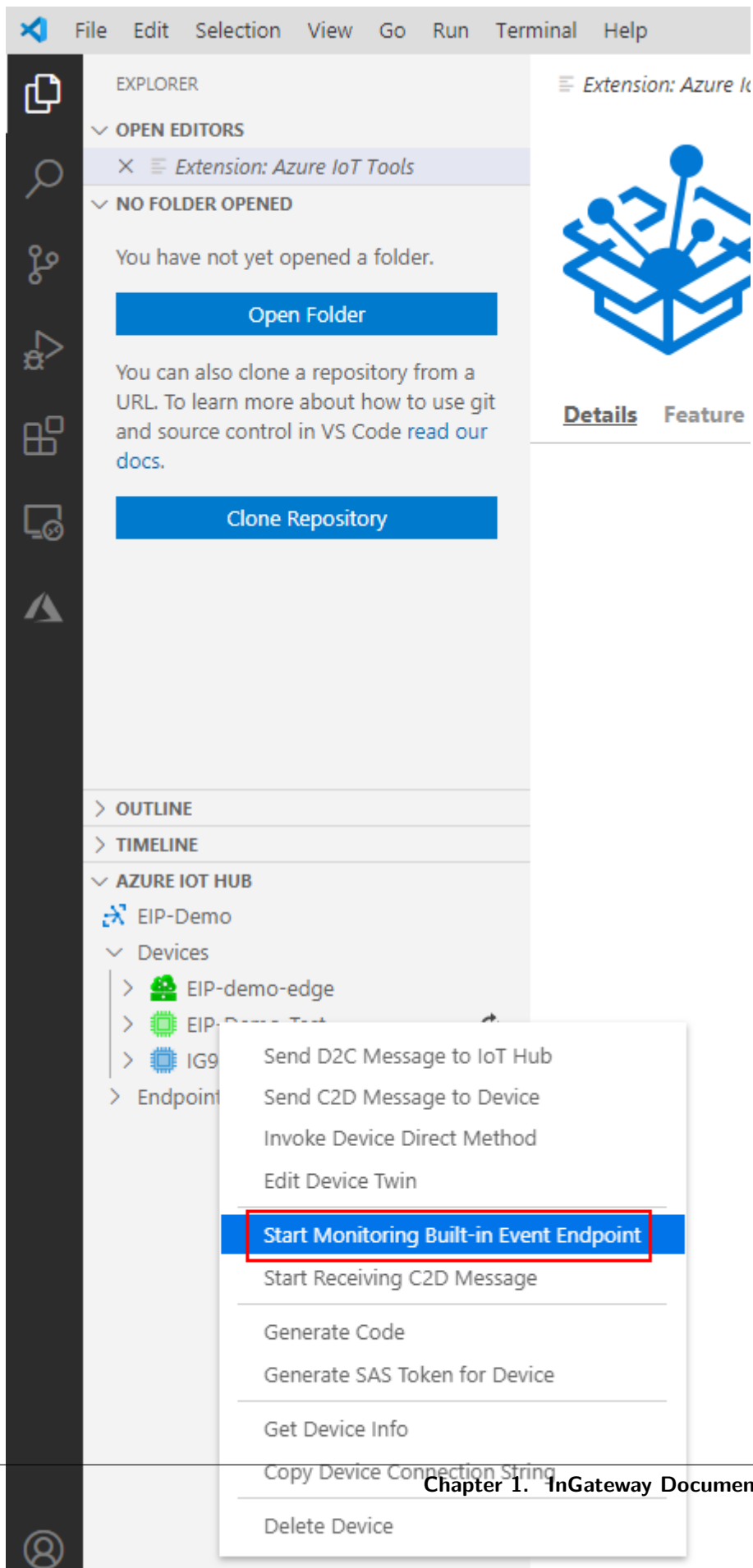
The bottom screenshot shows the Visual Studio Code interface. The "Azure IoT Tools" extension is installed and active. A dialog box prompts for the "IoT Hub Connection String (Press 'Enter' to confirm or 'Escape' to cancel)". The connection string from the top screenshot is pasted into the text box, which is also highlighted with a red box.

Enter the connection string and press Enter. The created Azure IoT device and its connection status are displayed under **AZURE IOT HUB**.

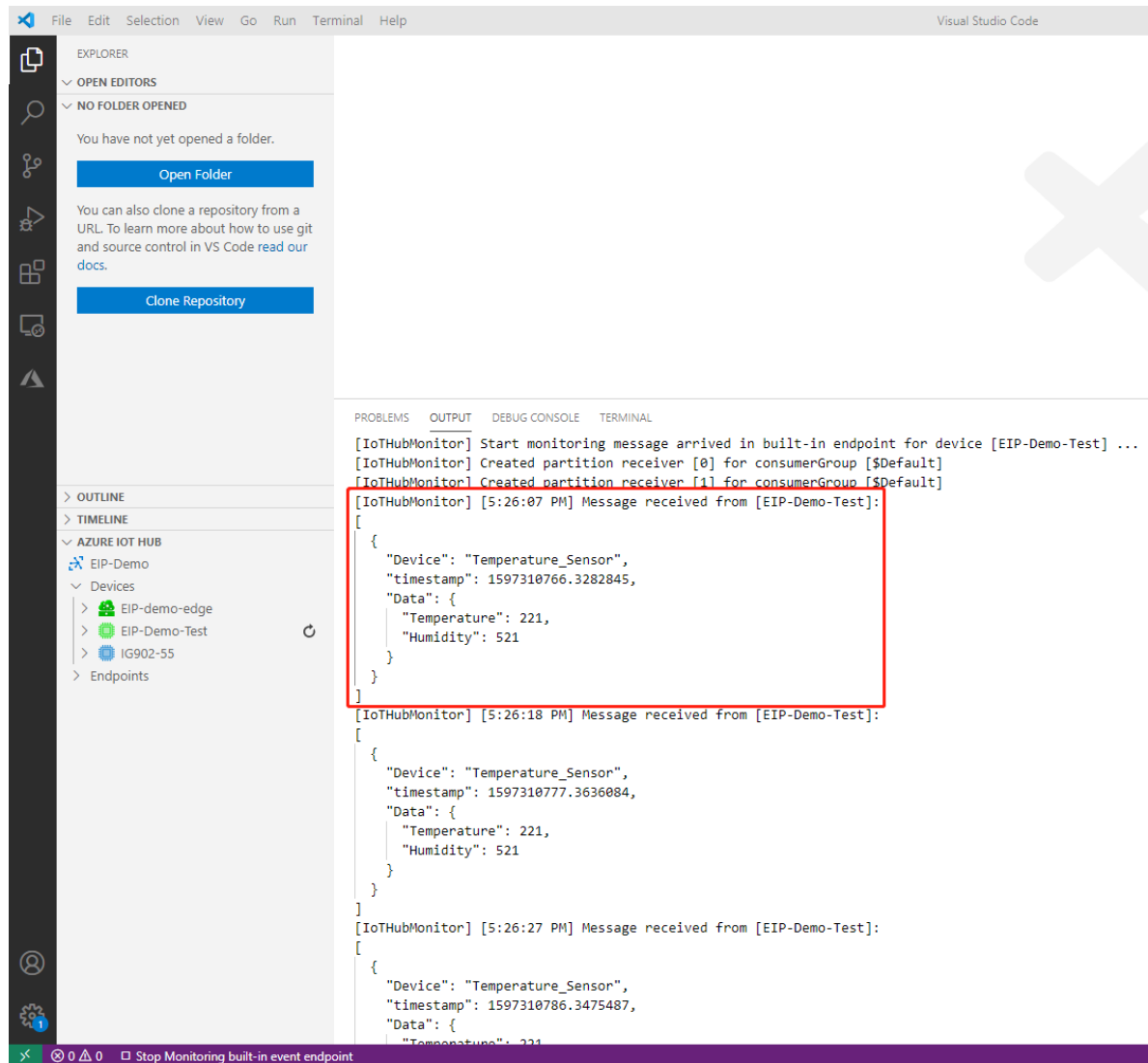


- Step 3: View the messages that IG902 uploads to the Azure IoT.

Right-click the Azure IoT device and choose **Start Monitoring Built-in Event Endpoint** to view the data that IG902 pushes to the IoT Hub.



The message content received by the IoT Hub is displayed on the **OUTPUT** tab page.



## 2.3 Subscribing to Azure IoT Messages

- Step 1: Configure the message for subscription.

Choose **Cloud > Message Management** and add the message for subscription. The following figure shows the configuration:

Overview / Edge Computing / Device Supervisor / Cloud

Subscribe Back

\* Name:

\* Main Function:  Should be the same with the name of the entry function in the script

\* Script:

```

1 import logging
2 import json
3 def ctl_test(message, wizard_api):
4     logging.info(message.data)
5     payload = json.loads(message.data)
6     if payload["method"] == "setValue":
7         message = {payload["TagName"]:payload["TagValue"]}
8         wizard_api.write_plc_values(message)

```

```

import logging
import json
def ctl_test(message, wizard_api):
    logging.info(message.data) # Print the subscription data. Assume that the
    payload data is {"method": "setValue", "TagName": "SP1", "TagValue": 12.3}.
    payload = json.loads(message.data) # Deserialize subscription data.
    if payload["method"] == "setValue": # Check whether the data is written.
        message = {payload["TagName"]:payload["TagValue"]} # Define the message to
        be delivered, including the variable names and values to be delivered.
        wizard_api.write_plc_values(message) # Call the write_plc_values method of
        the wizard_api module to deliver data from the message dictionary to the
        specified variable.

```

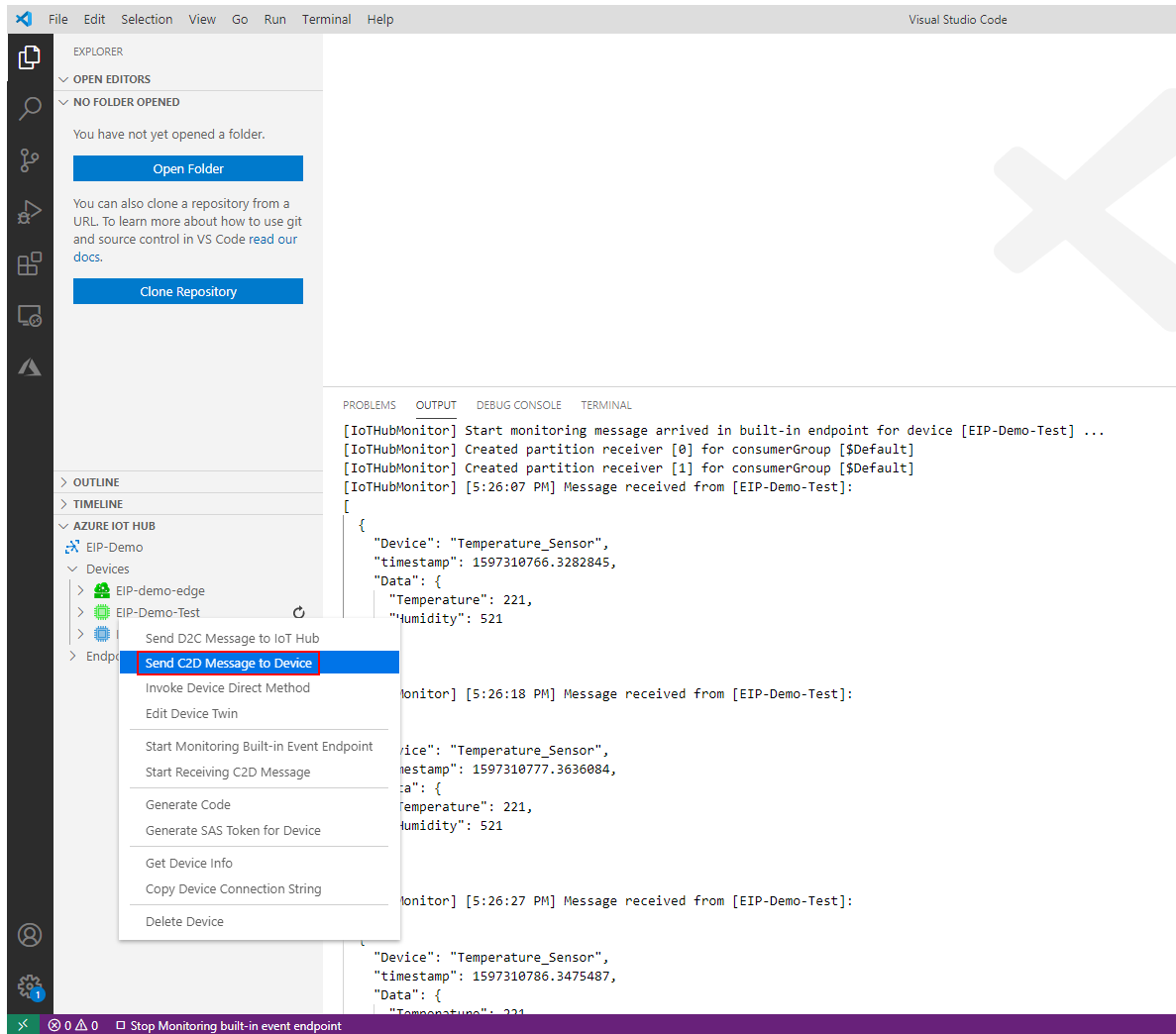
The message subscription parameters are described as follows:

- **Name:** custom subscription name.
- **Main Function:** name of the main function (entry function), which must be consistent with that in the script.
- **Script:** uses Python code to customize the packaging and processing logic. The main function parameters are as follows:
  - \* **Parameter 1:** message class sent by Azure IoT. The data and custom\_properties methods are supported. For details, see *Example of Subscribing to Azure IoT Messages*.
  - \* **Parameter 2:** Azure IoT API of the Device Supervisor. For details, see *Device Supervisor Azure IoT API Description*.
- Step 2: Use the Azure IoT Tools plug-in of the Visual Studio Code (VS Code for short) to establish the connection to the IoT Hub.

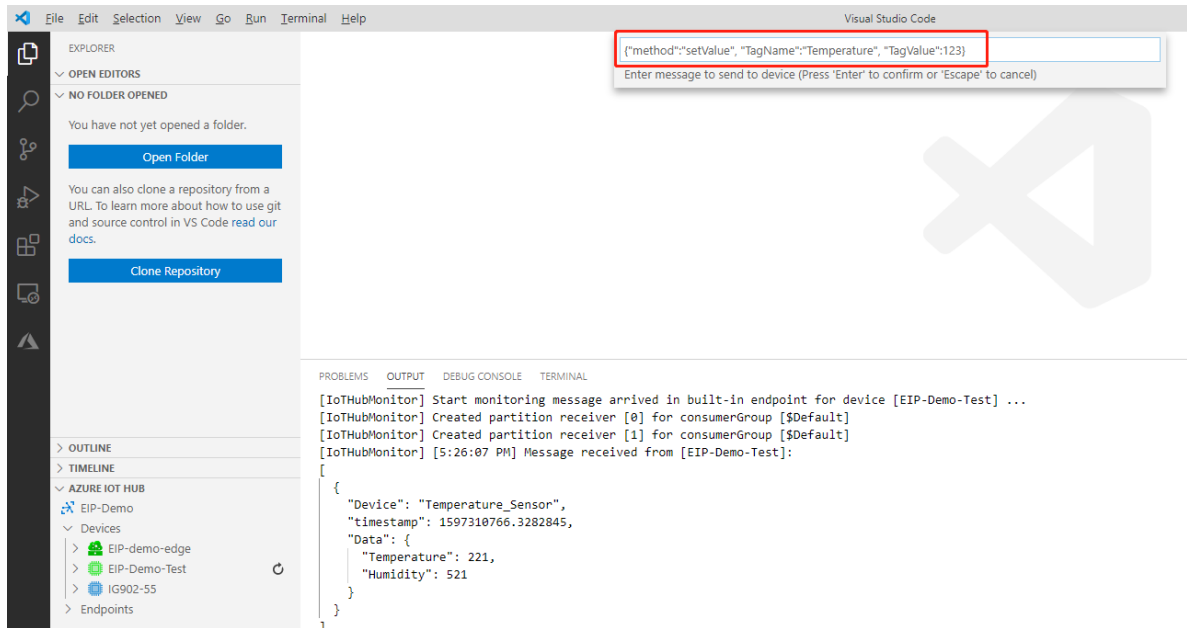
Same as Step 2 of “Publishing Messages to the Azure IoT” .

- Step 3: Use the Azure IoT Tools to send data to IG902.

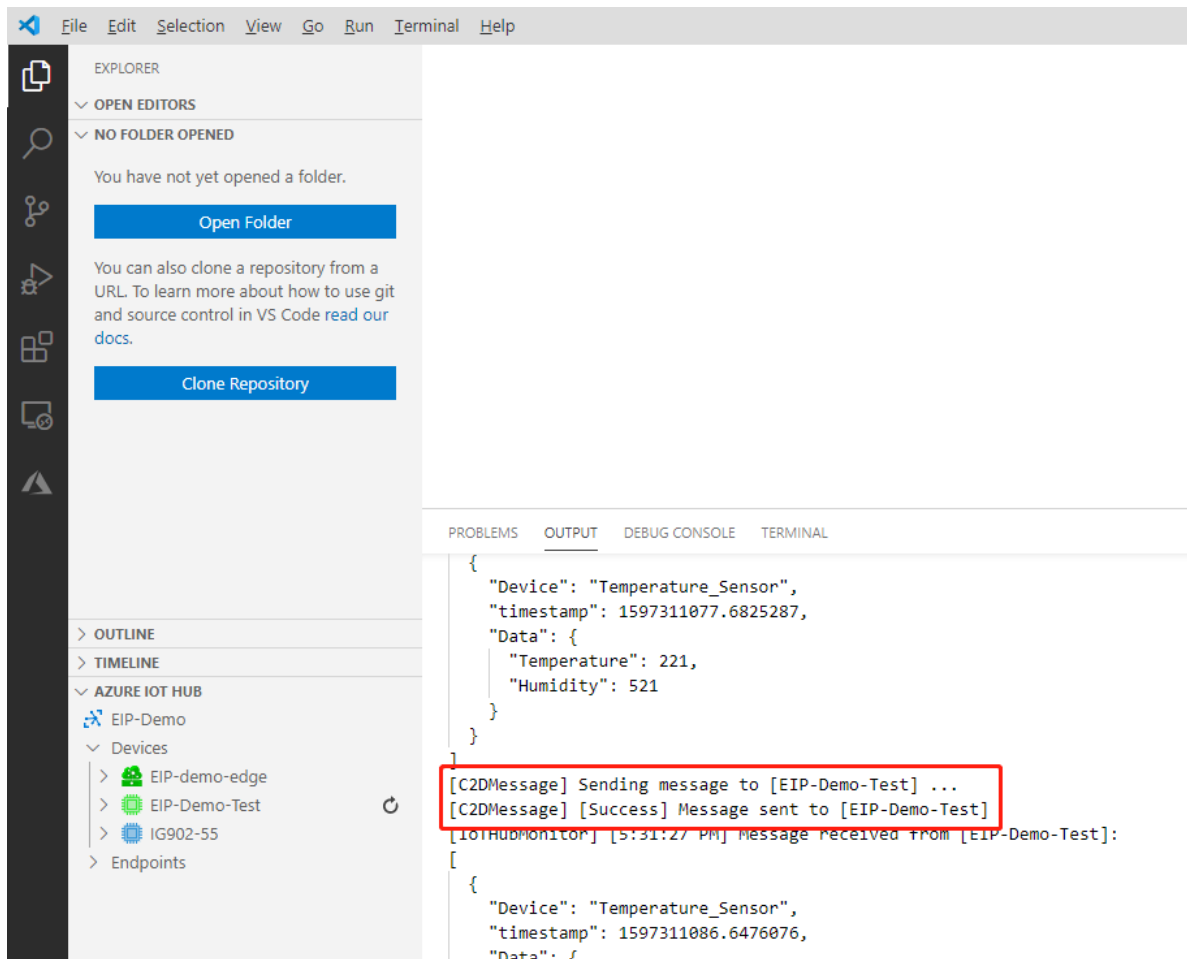
Right-click the Azure IoT device and choose **Send C2D Message to Device** to send the data to IG902.



Enter the data to be sent, for example, `{"method": "setValue", "TagName": "Temperature", "TagValue": 123}`.



Press Enter to send the data. Then, the success log is displayed on the **OUTPUT** tab page, and the value of the **Temperature** variable is changed to 123 on the **Device List** page.





概述 / 边缘计算 / 设备监控 / 设备列表

### 设备列表

操作: [+](#) [↓](#) [↑](#) [🗑](#)

☐ Temperature\_Sensor [🗑](#)  
 ModbusTCP  
 IP: 10.5.16.82 [✎](#)

共1项 [<](#) [1](#) [>](#)

### 变量列表(Temperature\_Sensor)

请输入变量名称 [🔍](#) 操作: [↓](#) [↑](#)

<input type="checkbox"/>	名称	分组	数据类型	地址	数值	描述	时间	操作 <a href="#">+</a>
<input type="checkbox"/>	Temperature	default	WORD	40001	123 °C <a href="#">✎</a>		2020-07-31 19:04:57	<a href="#">✎</a> <a href="#">🗑</a>
<input type="checkbox"/>	Humidity	default	WORD	40002	521 %RH		2020-07-31 19:04:57	<a href="#">✎</a> <a href="#">🗑</a>

[+ 添加到组](#) [🗑 删除](#) 共2项 [<](#) [1](#) [>](#) 50 条/页 [v](#)

Service data submitting and configuration data delivery between the Device Supervisor and the Azure IoT are completed.

## 1.3.4 Appendix

- *Example of Publishing Messages to the Azure IoT*
- *Example of Subscribing to Azure IoT Messages*
- *Device Supervisor Azure IoT API Description*

## Example of Publishing Messages to the Azure IoT

- Publication example 1: using return to publish user data and property data

Overview / Edge Computing / Device Supervisor / Cloud

### Publish

[Back](#)

• Name:

Group Type: ☒ Collect ☐ Alarm

• Group:

• Main Function:  Should be the same with the name of the entry function in the script

• Script:

```

1 import logging
2 import json
3
4
5 def vars_upload_test(data_collect, wizard_api):
6     value_list = []
7     for device, val_dict in data_collect['values'].items():
8         value_dict = {
9             "Device": device,
10            "timestamp": data_collect["timestamp"],
11            "Data": {}
12        }
13        for id, val in val_dict.items():
14            value_dict["Data"][id] = val["raw_data"]
15        value_list.append(value_dict)
16        logging.info(value_list)
17        upload_data = {"data": json.dumps(value_list), "custom_properties": {"Name": "properties upload"}}
18        return(upload_data)

```

[Submit](#) [Reset](#)

```

import logging
import json
"""
Logs are generally generated in the gateway in the following ways:
1. import logging: uses logging.info(XXX) to generate logs. Display of logs
↳ generated in this way is not controlled by the log level parameter on the global
↳ parameter page.
2. from common.Logger import logger: uses logger.info(XXX) to generate logs.
↳ Display of logs generated in this way is controlled by the log level parameter on
↳ the global parameter page.
"""

def vars_upload_test(data_collect, wizard_api): # Define the main publishing
↳ function.
    value_list = [] # Define the data list.
    for device, val_dict in data_collect['values'].items(): # Traverse the values
↳ dictionary. The dictionary contains the device name and the variables of the
↳ device.
        value_dict = { # Customize the user data dictionary.
            "Device": device,
            "timestamp": data_collect["timestamp"],
            "Data": {}
        }
        for id, val in val_dict.items(): # Traverse variables and assign values for
↳ the Data dictionary.
            value_dict["Data"][id] = val["raw_data"]
            value_list.append(value_dict) # Add data in value_dict to value_list in
↳ sequence.
        logging.info(value_list) # Print data in value_list in app logs in the
↳ following format: [{'Device': 'S7-1200', 'timestamp': 1589538347.5604711, 'Data':
↳ {'Test1': False, 'Test2': 12}}].
        upload_data = {"data": json.dumps(value_list), "custom_properties": {"Name":
↳ "properties upload"}} # Define submitted data with the data type of dictionary.
↳ The user data is the value of "data", the data type is string, the property data
↳ is the value of "custom_properties", and the data type is dictionary.
        return(upload_data) # Send upload_data to the app, which then uploads it to the
↳ Azure IoT by collection time. If it fails to be sent, the data is cached and
↳ uploaded to the Azure IoT by collection time after the connection recovers.

```

- Publication example 2: using `send_message_to_cloud` to send user data and using `save_data` to store the variables that fail to upload

Overview / Edge Computing / Device Supervisor / Cloud

Publish Back

• Name:

Group Type: ☒ Collect ☐ Alarm

• Group:  ✕

• Main Function:  ⚠ Should be the same with the name of the entry function in the script

• Script:

```

1 import logging
2 import json
3
4 def vars_upload_test(data_collect, wizard_api):
5     value_list = []
6     for device, val_dict in data_collect['values'].items():
7         value_dict = {
8             "Device": device,
9             "timestamp": data_collect["timestamp"],
10            "Data": {}
11        }
12        for id, val in val_dict.items():
13            value_dict["Data"][id] = val["raw_data"]
14        value_list.append(value_dict)
15        logging.info(value_list)
16        if not wizard_api.send_message_to_cloud(json.dumps(value_list)):
17            wizard_api.save_data(json.dumps(value_list), 'default')

```

```
import logging
```

```
import json
```

```
"""
```

*Logs are generally generated in the gateway in the following ways:*

*1. import logging: uses logging.info(XXX) to generate logs. Display of logs*

*↳ generated in this way is not controlled by the log level parameter on the global*  
*↳ parameter page.*

*2. from common.Logger import logger: uses logger.info(XXX) to generate logs.*

*↳ Display of logs generated in this way is controlled by the log level parameter on*  
*↳ the global parameter page.*

```
"""
```

```
def vars_upload_test(data_collect, wizard_api): # Define the main publishing
```

*↳ function.*

```
    value_list = [] # Define the data list.
```

```
    for device, val_dict in data_collect['values'].items(): # Traverse the values
```

*↳ dictionary. The dictionary contains the device name and the variables of the*  
*↳ device.*

```
        value_dict = { # Customize the user data dictionary.
```

```
            "Device": device,
```

```
            "timestamp": data_collect["timestamp"],
```

```
            "Data": {}
```

```
        }
```

```
        for id, val in val_dict.items(): # Traverse variables and assign values for
```

*↳ the Data dictionary.*

```
            value_dict["Data"][id] = val["raw_data"]
```

```
        value_list.append(value_dict) # Add data in value_dict to value_list in
```

*↳ sequence.*

(continues on next page)

(continued from previous page)

```

logging.info(value_list) # Print data in value_list in app logs in the
↳ following format: [{'Device': 'S7-1200', 'timestamp': 1589538347.5604711, 'Data':
↳ {'Test1': False, 'Test2': 12}}].

if not wizard_api.send_message_to_cloud(json.dumps(value_list)): # Call the
↳ send_message_to_cloud method of the wizard_api module to send value_list to the
↳ Azure IoT and check whether it is successful.

wizard_api.save_data(json.dumps(value_list), 'default') # If it fails to be
↳ sent, the data is stored and uploaded by time after the connection recovers.

```

## Example of Subscribing to Azure IoT Messages

- Subscribing to user data

The following is a configuration example:

Overview / Edge Computing / Device Supervisor / Cloud

Subscribe Back

• Name:

• Main Function:  Should be the same with the name of the entry function in the script

• Script:

```

1 import logging
2 import json
3 def ctl_test(message, wizard_api):
4     logging.info(message.data)
5     if payload["method"] == "setValue":
6         message = {payload["TagName"]:payload["TagValue"]}
7         wizard_api.write_plc_values(message)

```

The script is as follows:

```

import logging
import json
def ctl_test(message, wizard_api):
    logging.info(message.data) # Print the subscription data. Assume that the
↳ payload data is {"method": "setValue", "TagName": "SP1", "TagValue": 12.3}.
    payload = json.loads(message.data) # Deserialize subscription data.
    if payload["method"] == "setValue": # Check whether the data is written.
        message = {payload["TagName"]:payload["TagValue"]} # Define the message to
↳ be delivered, including the variable names and values to be delivered.
        wizard_api.write_plc_values(message) # Call the write_plc_values method of
↳ the wizard_api module to deliver data from the message dictionary to the
↳ specified variable.

```

- Subscribing to property data

The following is a configuration example:

Overview / Edge Computing / Device Supervisor / Cloud

---

Subscribe Back

• Name:

• Main Function:  ⓘ Should be the same with the name of the entry function in the script

• Script:

```

1 import logging
2 import json
3 def ctl_test(message, wizard_api):
4     logging.info(message.custom_properties)

```

The script is as follows:

```

import logging
import json
def ctl_test(message, wizard_api):
    logging.info(message.custom_properties) # Print the subscription data.

```

## Device Supervisor Azure IoT API Description

For details about the basic `wizard_api` configuration, see [Device Supervisor API Description](#). (Note: The data can be stored only through the group name. For details, see *Publication example 2*). If the cloud service type is Azure IoT, `wizard_api` additionally provides the following method:

- `send_message_to_cloud`
  - Method Description: data submitting method.
  - Parameter
    - \* `data`: user data to be submitted. This parameter cannot be left empty. The data type must be string, and the size of data submitted at a time cannot exceed 256 KB.
    - \* `custom_properties`: property data to be submitted. The data type of property data must be dictionary, and the data type of values in the dictionary must be integer, floating-point number, or string. The size of data submitted at a time cannot exceed 81 KB.
  - Usage example
    - \* Submitting user data

Overview / Edge Computing / Device Supervisor / Cloud

Publish Back

Name:

Group Type: ☒ Collect ☐ Alarm

Group:

Main Function:  Should be the same with the name of the entry function in the script

Script:

```

1 import logging
2 import json
3
4 def vars_upload_test(data_collect, wizard_api):
5     value_list = []
6     for device, val_dict in data_collect['values'].items():
7         value_dict = {
8             "Device": device,
9             "timestamp": data_collect["timestamp"],
10            "Data": {}
11        }
12        for id, val in val_dict.items():
13            value_dict["Data"][id] = val["raw_data"]
14        value_list.append(value_dict)
15        logging.info(value_list)
16        wizard_api.send_message_to_cloud(json.dumps(value_list))

```

```
import logging
```

```
import json
```

```
"""
```

Logs are generally generated in the gateway in the following ways:

1. `import logging`: uses `logging.info(XXX)` to generate logs. Display of logs

→ generated in this way is not controlled by the log level parameter on the

→ global parameter page.

2. `from common.Logger import logger`: uses `logger.info(XXX)` to generate logs.

→ Display of logs generated in this way is controlled by the log level

→ parameter on the global parameter page.

```
"""
```

```
def vars_upload_test(data_collect, wizard_api): # Define the main
```

→ publishing function.

```
    value_list = [] # Define the data list.
```

```
    for device, val_dict in data_collect['values'].items(): # Traverse the
```

→ values dictionary. The dictionary contains the device name and the

→ variables of the device.

```
        value_dict = { # Customize the user data dictionary.
```

```
            "Device": device,
```

```
            "timestamp": data_collect["timestamp"],
```

```
            "Data": {}
```

```
        }
```

```
        for id, val in val_dict.items(): # Traverse variables and assign
```

→ values for the Data dictionary.

```
            value_dict["Data"][id] = val["raw_data"]
```

```
        value_list.append(value_dict) # Add data in value_dict to value_
```

→ list in sequence.

(continues on next page)

(continued from previous page)

```

logging.info(value_list) # Print data in value_list in app logs in the
↳ following format: [{'Device': 'S7-1200', 'timestamp': 1589538347.5604711,
↳ 'Data': {'Test1': False, 'Test2': 12}}].

wizard_api.send_message_to_cloud(json.dumps(value_list)) # Call the
↳ send_message_to_cloud method of the wizard_api module to send value_list
↳ (user data) to the Azure IoT.

```

### \* Submitting property data

Overview / Edge Computing / Device Supervisor / Cloud

Publish Back

Name:

Group Type: ☒ Collect ☐ Alarm

Group:  ✕

Main Function:  Should be the same with the name of the entry function in the script

Script:

```

1 import logging
2 import json
3
4
5 def vars_upload_test(data_collect, wizard_api):
6     value_dict = {"Name": "properties upload"}
7     wizard_api.send_message_to_cloud("properties", value_dict)

```

```

import logging
import json
"""
Logs are generally generated in the gateway in the following ways:
1. import logging: uses logging.info(XXX) to generate logs. Display of logs
↳ generated in this way is not controlled by the log level parameter on the
↳ global parameter page.
2. from common.Logger import logger: uses logger.info(XXX) to generate logs.
↳ Display of logs generated in this way is controlled by the log level
↳ parameter on the global parameter page.
"""

def vars_upload_test(data_collect, wizard_api): # Define the main
↳ publishing function.
    value_dict = {"Name": "properties upload"} # Customize the property data
↳ dictionary.
    wizard_api.send_message_to_cloud("properties", value_dict) # Call the
↳ send_message_to_cloud method of the wizard_api module to send value_dict
↳ (property data) to the Azure IoT. Note: When property data is sent,
↳ parameter 1 cannot be left empty, and the data type must be string

```

(continues on next page)

(continued from previous page)

## 1.3.5 FAQ

### Q1: The Azure IoT Connection Frequently Fails Shortly After It Is Established

A1: The run logs of the app contain Paho returned rc==1.

```
[2020-08-10 14:55:48.622] [INFO] [pipeline_stages_mqtt.py 274]: _on_mqtt_connected called
[2020-08-10 14:55:48.624] [DEBUG] [pipeline_stages_base.py 253]: PipelineRootStage: ConnectedEvent received. Calling on_connected_handler
[2020-08-10 14:55:48.626] [DEBUG] [pipeline_thread.py 108]: Starting _on_connected in callback thread
[2020-08-10 14:55:48.634] [ERROR] [handle_exceptions.py 28]: Exception caught in background thread. Unable to handle.
[2020-08-10 14:55:48.636] [ERROR] [handle_exceptions.py 29]: ["TypeError: 'NoneType' object is not callable\n"]
[2020-08-10 14:55:48.641] [DEBUG] [pipeline_stages_mqtt.py 280]: completing connect op
[2020-08-10 14:55:48.646] [DEBUG] [pipeline_stages_mqtt.py 103]: MQTTTransportStage(ConnectOperation): cancelling watchdog
[2020-08-10 14:55:48.660] [DEBUG] [pipeline_ops_base.py 109]: ConnectOperation: completing without error
[2020-08-10 14:55:48.664] [DEBUG] [pipeline_stages_base.py 525]: ConnectionLockStage(ConnectOperation): op succeeded. Unblocking queue
[2020-08-10 14:55:48.665] [DEBUG] [pipeline_stages_base.py 550]: ConnectionLockStage(ConnectOperation): unblocking and releasing queued ops.
[2020-08-10 14:55:48.667] [INFO] [pipeline_stages_base.py 553]: ConnectionLockStage(ConnectOperation): processing 0 items in queue
[2020-08-10 14:55:48.662] [DEBUG] [pipeline_stages_base.py 1046]: ReconnectStage(ConnectOperation): on_connect_complete error=None state=LOGICALLY_CONNECTED never_connected=False connected=True
[2020-08-10 14:55:48.666] [INFO] [pipeline_stages_base.py 1141]: ReconnectStage: completing waiting ops with error=None
[2020-08-10 14:55:48.670] [DEBUG] [pipeline_stages_base.py 1095]: ReconnectStage: Reconnect timer expired. State is WAITING_TO_RECONNECT Connected is False.
[2020-08-10 14:55:48.674] [DEBUG] [pipeline_stages_base.py 1071]: ReconnectStage: sending new connect op down
[2020-08-10 14:55:48.678] [DEBUG] [pipeline_stages_base.py 541]: ConnectionLockStage(ConnectOperation): blocking
[2020-08-10 14:55:48.682] [INFO] [pipeline_stages_mqtt.py 170]: MQTTTransportStage(ConnectOperation): connecting
[2020-08-10 14:55:48.686] [DEBUG] [pipeline_stages_mqtt.py 68]: MQTTTransportStage(ConnectOperation): Starting watchdog
[2020-08-10 14:55:48.691] [INFO] [mqtt_transport.py 376]: connecting to mqtt broker
[2020-08-10 14:55:48.694] [INFO] [mqtt_transport.py 387]: Connect using port 8883 (TCP)
[2020-08-10 14:55:48.900] [DEBUG] [client.py 2165]: Sending CONNECT (ul, pl, wr0, wq0, wf0, c0, k60) client_id='IG902-55'
[2020-08-10 14:55:48.909] [DEBUG] [mqtt_transport.py 428]: _mqtt_client.connect returned rc=0
[2020-08-10 14:55:48.915] [INFO] [pipeline_stages_mqtt.py 321]: MQTTTransportStage: _on_mqtt_disconnect called: ConnectionDroppedError('Paho returned rc==1')
[2020-08-10 14:55:48.919] [DEBUG] [pipeline_stages_base.py 1002]: ReconnectStage(DisconnectedEvent): State is WAITING_TO_RECONNECT Connected is False.
[2020-08-10 14:55:48.923] [DEBUG] [pipeline_stages_base.py 261]: PipelineRootStage: DisconnectedEvent received. Calling on_disconnected_handler
[2020-08-10 14:55:48.924] [DEBUG] [pipeline_thread.py 108]: Starting _on_disconnected in callback thread
[2020-08-10 14:55:48.929] [INFO] [pipeline_stages_mqtt.py 359]: MQTTTransportStage: disconnection was unexpected
[2020-08-10 14:55:48.931] [INFO] [sync_clients.py 90]: Connection State - Disconnected
[2020-08-10 14:55:48.936] [INFO] [sync_clients.py 92]: Cleared all pending method requests due to disconnect
[2020-08-10 14:55:48.935] [INFO] [handle_exceptions.py 52]: Unexpected disconnection. Safe to ignore since other stages will reconnect.
[2020-08-10 14:55:48.942] [INFO] [handle_exceptions.py 53]: azure.iot.device.common.transport.exceptions.ConnectionDroppedError: ConnectionDroppedError('Paho returned rc==1')
```

The above exception was the direct cause of the following exception:

Traceback (most recent call last):  
 File "/var/user/app/device\_supervisorbak/device\_supervisor/lib/azure/iot/device/common/handle\_exceptions.py", line 43, in swallow\_unraised\_exception  
 azure.iot.device.common.transport.exceptions.ConnectionDroppedError: ConnectionDroppedError(None) caused by ConnectionDroppedError('Paho returned rc==1')

Log in to the Azure IoT, select the IoT devices related to the connection string, and choose **IoT Hub > IoT devices**. The message “You’ve reached your daily message quota, so you won’t be able to send new message or view device list” is displayed. If this occurs, wait until the daily message quota is available again, or increase the daily message quota.

The screenshot shows the Microsoft Azure portal interface for IoT Hub. The top navigation bar includes the Microsoft Azure logo and a search bar. The main content area displays the 'EIP-Demo | IoT devices' page. A red-bordered error message is prominently displayed at the top of the page, stating: "You've reached your daily message quota, so you won't be able to send new message or view device list. Because you're using the free edition of IoT Hub, you can't increase message quota. Wait until 12 AM UTC or create a hub using a paid plan. Learn more." Below the error message, the 'View, create, delete, and update devices in your IoT Hub.' section is visible, featuring a table with columns: DEVICE ID, STATUS, LAST STATUS UPDATE (...), AUTHENTICATION TYPE, and CLOUD TO DEVICE MESSAGE COUNT. The table is currently empty.



## 1.4 DeviceSupervisor 2.0 Upgrade Notes

The DeviceSupervisor version **1.2.X** is referred to as **DS 1.0** and the DeviceSupervisor version **2.X.X** is referred to as **DS 2.0**. When upgrading from DS 1.0 to DS 2.0, you need to pay attention to the following points.

### 1.4.1 Global description

1. Only support the smooth upgrade of DS 1.0 version 1.2.9 and above to DS 2.0;
2. After the upgrade, the configuration information of DS 1.0 will be translated into the configuration of DS 2.0, and it cannot be rolled back to the DS 1.0 version after the upgrade;
3. After smooth upgrade, the configuration format obtained by calling DS 1.0 **get\_tag\_config** API has changed;
4. The configuration of DS 2.0 is inconsistent with DS 1.0, and the configuration file of DS 1.0 cannot be imported into DS 2.0.

### 1.4.2 Explanation of terms after upgrade

- Device -> Controller
- Variable -> Measuring point
- Alarm Strategy -> Alarm Rules
- Group (Polling interval) -> Group (Reporting interval)
- Added controller “Polling Interval”

### 1.4.3 Update instructions for each functional module

#### Measure Monitor

1. After the upgrade, the configuration of **OPCUA/EtherNetIP** measure point will be discarded;
2. The data type **BOOL** will be converted to the **BIT** type;
3. DS 2.0 no longer supports configuring multiple devices (controllers) with the same IP address and port;
4. DS 1.0's **write-only** mode variables will become **read/write** mode after upgrade;
5. After the smooth upgrade, the modbus address changes: **20000->110000**, **40000->310000**, **50000->410000**;
6. After the smooth upgrade, the measurement point upload mode, realtime -> periodic.

## Alarm

1. After the upgrade, the history alarms and offline cache data stored in DS 1.0 will be cleared;
2. The alarm strategy in DS 2.0 no longer supports the “direct use of address” strategy, and the alarm strategy using this method will become invalid after the upgrade;
3. After the upgrade, the alarm group of DS 1.0 is deleted. DS 2.0 only distinguishes different alarms according to the alarm name. It should be noted that if the DS 1.0 cloud service script refers to an empty alarm group, the script may not be available after the upgrade because there is no alarm group. To run, please manually select the trigger source type in the cloud service script.

## Cloud Service

1. After DS 1.0 is upgraded, **write\_plc\_values** API will no longer support anonymous device names to modify measure point values, that is, to modify PLC values, you need to specify the controller name. Therefore, the Alibaba Cloud attribute setting script of DS 1.0 will be invalid. You need to specify the name of the device to be modified (DS 2.0 is called the controller) in the script;
2. After upgrading the Alibaba Cloud custom RRPC script of DS 1.0, the topic response will be invalid, but the script function can be executed normally. It is recommended to use the DS 2.0 API to modify the script;
3. Smooth upgrade does not support GreenGrass Core related configuration migration;
4. After the smooth upgrade, the certificate name used by the cloud service will become the default certificate name of DS 2.0, which does not affect the use of the function.

## Parameter Settings

1. After the smooth upgrade, two new parameters, SN and MAC, are added to the custom parameters. The built-in parameter gateway\_sn in DS 1.0 will also be explicitly added. If gateway\_sn is used in the script, please delete gateway\_sn carefully. You can use the new DS 2.0 SN replaces gateway\_sn;
2. After the smooth upgrade, the parameter setting canceled the item of the **maximum number of historical data**.