ALIBABA CLOUD

Alibaba Cloud

物联网边缘计算

Best Practices

Document Version: 20211228

Alibaba Cloud

Legal disclaimer

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

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

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

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

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

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

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

Document conventions

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

Table of Contents

1.Connect a Modbus slave device to an edge instance over Modbus TCP

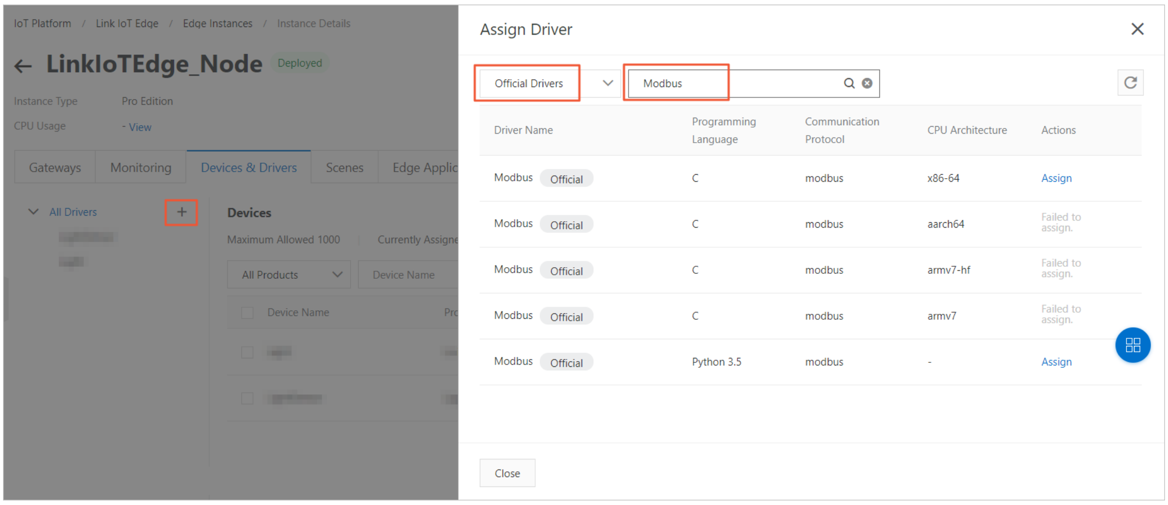
Modbus is a communication protocol based on the master/slave model. The master device requests data from slave devices. The Modbus driver of a gateway serves as the master device, and devices that connect to the gateway through the Modbus driver serve as slave devices. After the master device sends a query message to a slave device, the slave device sends a response message.

Preparations

1. A Ubuntu 16.04 x86_64 system is prepared to run a gateway.
2. A Windows host is prepared to run a Modbus slave simulator.
3. Download a Modbus slave simulator from the URL of Modbus tools and install the simulator on the Windows host.
4. Check the firewall settings of the Windows host and make sure that access to the Modbus slave device is allows through port 502. If the access is denied, disable the firewall or change firewall settings to allow access to the port.
5. Create an edge instance and enable a gateway. For more information, see Build an environment.

Step 1: Assign a Modbus driver to the edge instance

1. Log on to the Link IoT Edge console.
2. In the left-side navigation pane, click **Edge Instances**, and click **View** next to the required edge instance.
3. Assign the required Modbus driver to the edge instance based on the CPU of the related gateway. For more information, see Modbus drivers.



IoT Platform / Link IoT Edge / Edge Instances / Instance Details

LinkIoTEdge_Node Deployed

Instance Type Pro Edition

CPU Usage - View

Assign Driver

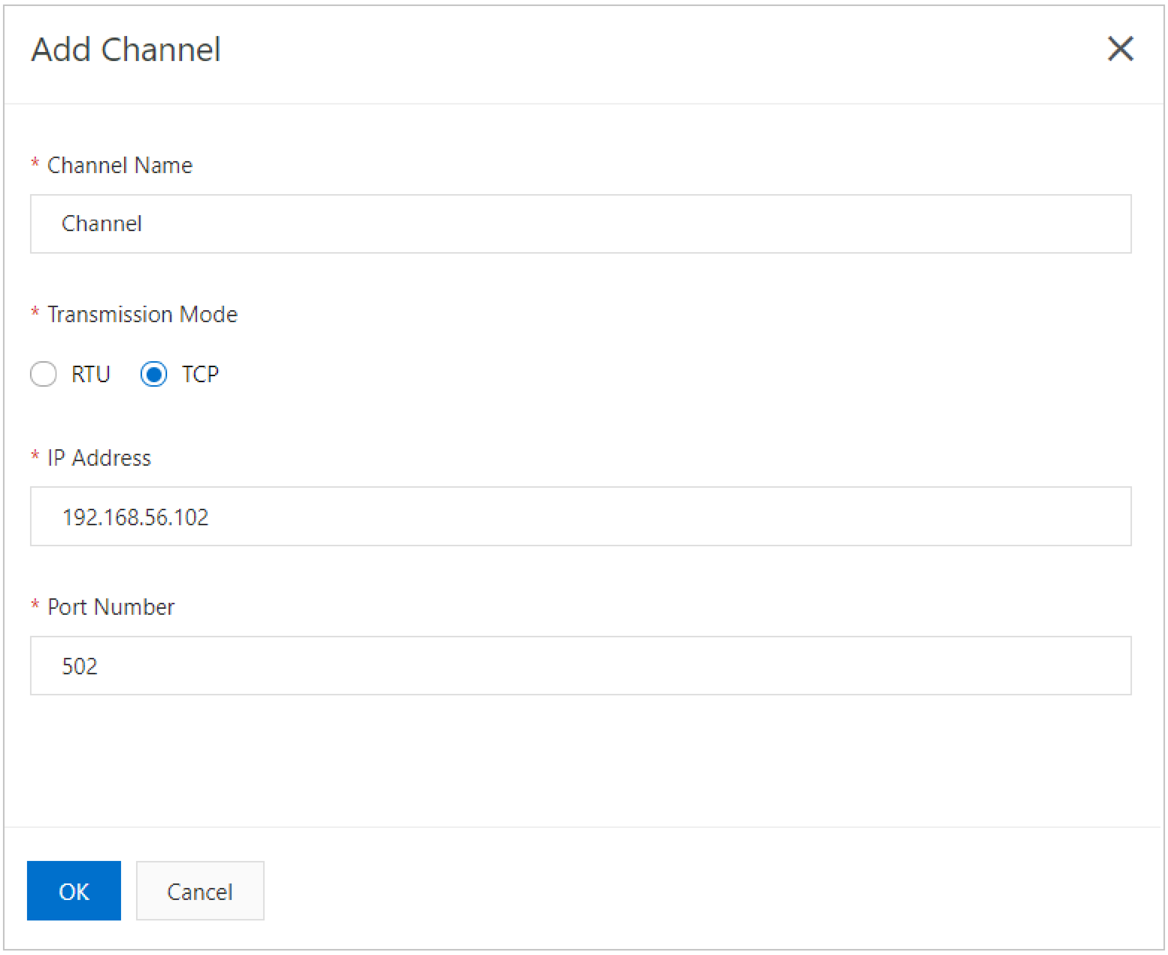
Official Drivers

Modbus

Driver Name	Programming Language	Communication Protocol	CPU Architecture	Actions
Modbus Official	C	modbus	x86-64	Assign
Modbus Official	C	modbus	aarch64	Failed to assign.
Modbus Official	C	modbus	armv7-hf	Failed to assign.
Modbus Official	C	modbus	armv7	Failed to assign.
Modbus Official	Python 3.5	modbus	-	Assign

4. On the Instance Details page of the edge instance, click the name of the assigned driver. Then, click **Driver Configurations** in the **Devices** section.
5. In the Driver Configurations panel, click **Add Channel**. In the Add Channel panel, set the required parameters to add a channel to the Modbus driver.

   For more information about the parameters, see Modbus driver configurations.



Add Channel

* Channel Name

Channel

* Transmission Mode

○ RTU ● TCP

* IP Address

192.168.56.102

* Port Number

502

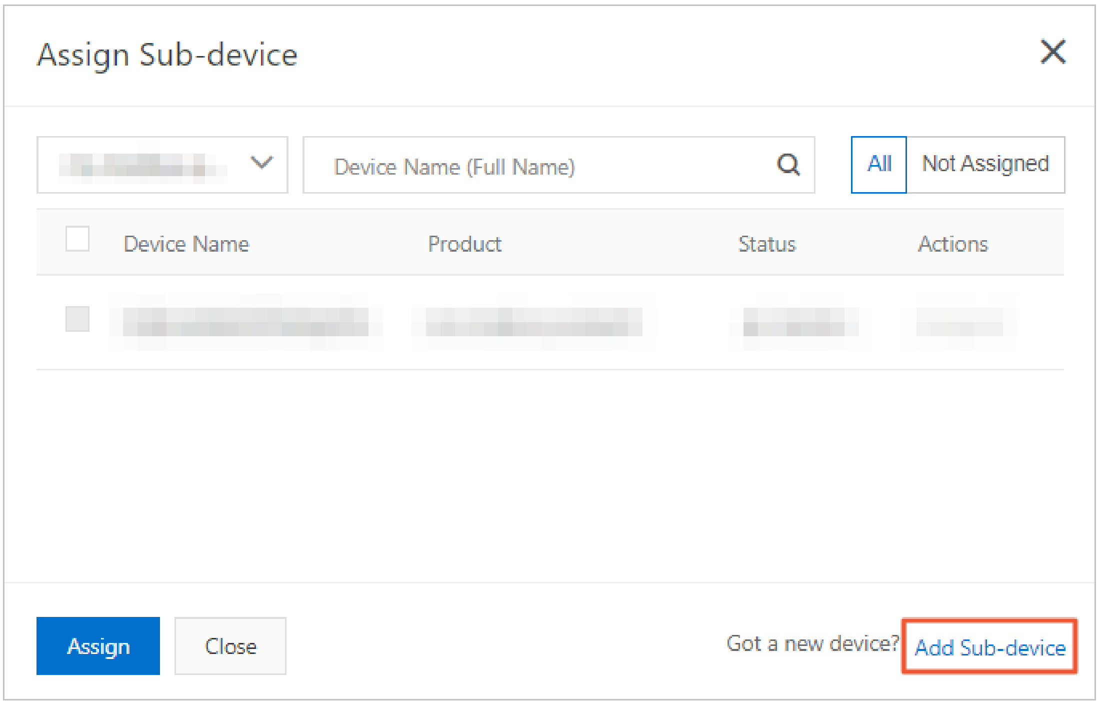
Step 2: Assign a sub-device to the Modbus driver

1. In the **Devices** section, click **Assign Sub-device**. In the Assign Sub-device panel, assign a sub-device to the edge instance.

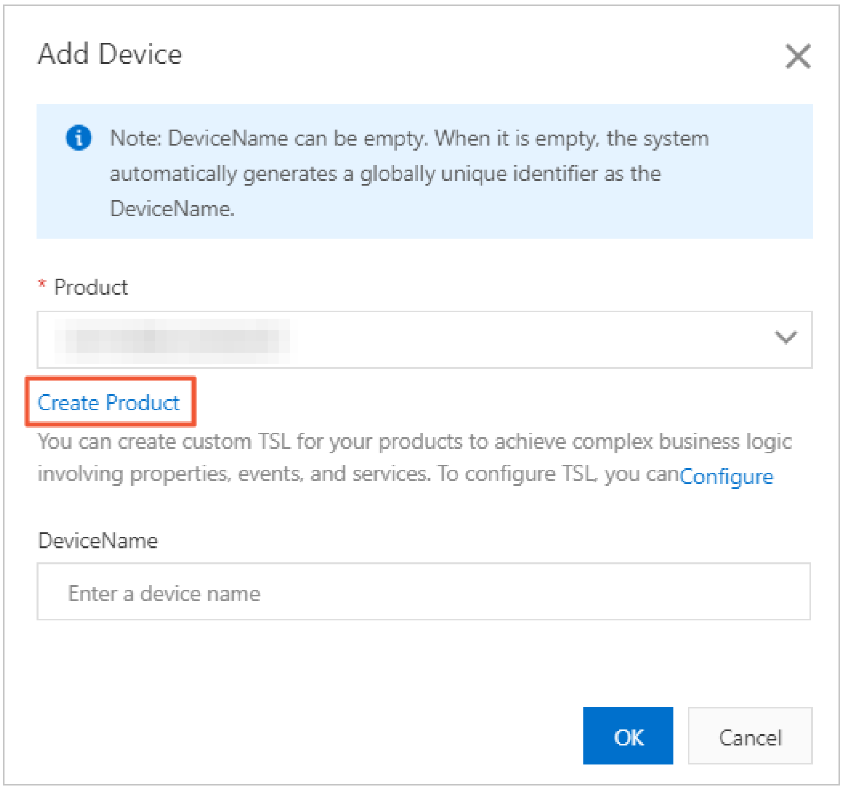
   You can select an existing Modbus device or create a sub-device. To create a sub-device, proceed with the following steps.

   > **Note** If you want to select an existing Modbus device, the product to which the device belongs must be connected to a gateway by using the Modbus protocol. For more information, see Create a product.

2. In the **Assign Sub-device** panel, click **Add Sub-device**.



3. In the **Add Device** dialog box, click **Create Product** and create a product to which the new Modbus device belongs.



Note: DeviceName can be empty. When it is empty, the system automatically generates a globally unique identifier as the DeviceName.

You can create custom TSL for your products to achieve complex business logic involving properties, events, and services. To configure TSL, you canConfigure

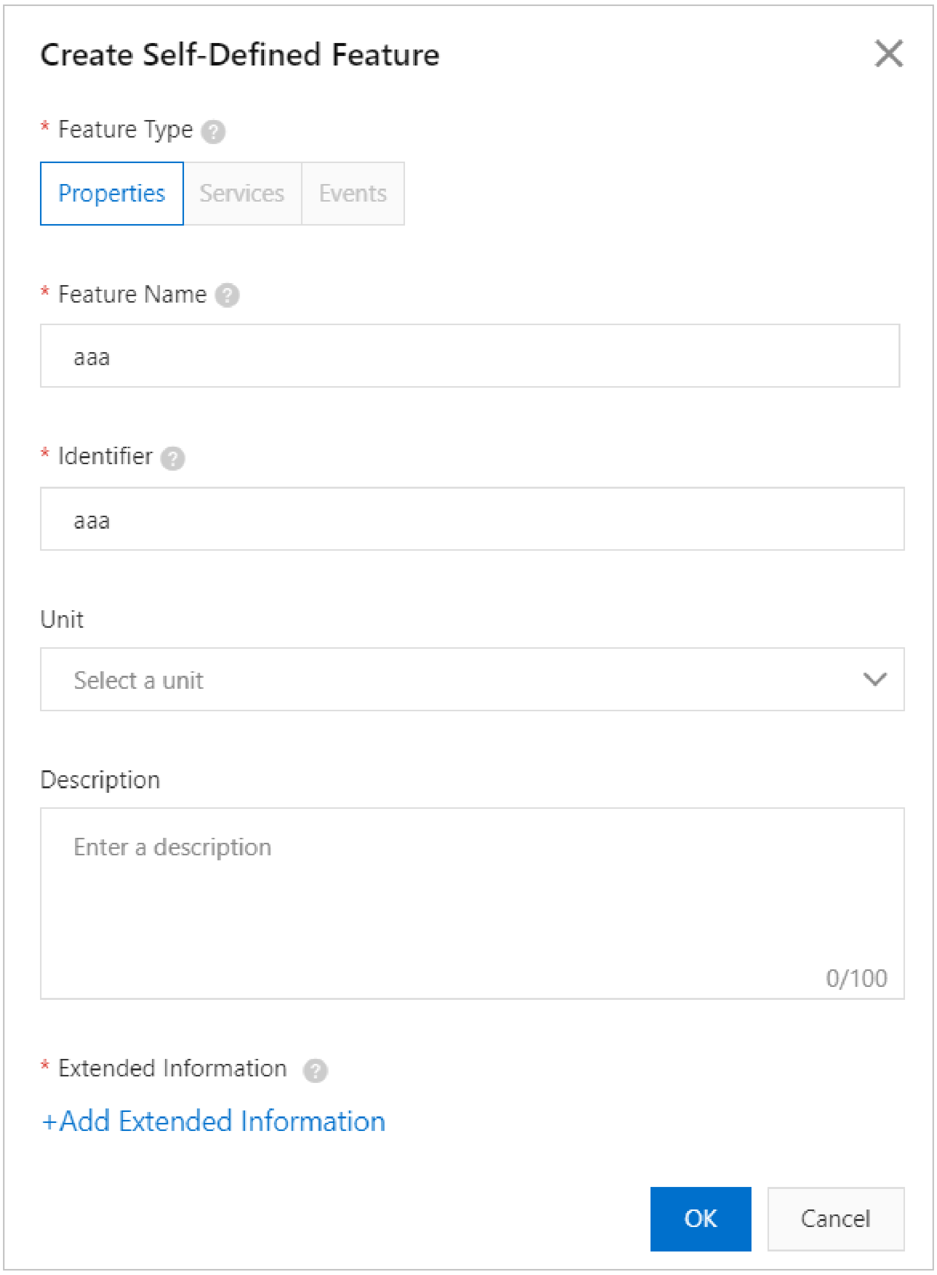
4. In the **Create Product** dialog box, set the parameters as required and click **OK**.

Parameter description

Parameter	Description

Parameter	Description
Product Name	The name of the product. The product name must be unique within the current Alibaba Cloud account. The name must be 4 to 30 characters in length and can contain letters, digits, underscores (_), hyphens (-), at signs (@), and parentheses ().
Gateway Connection Protocol	The communications protocol. You must set this parameter to Modbus.
Authentication Mode	The authentication method. Select an authentication method that is suitable for your devices. For more information, see Authenticate devices.
Product Description	The description of the product. This parameter is optional.

5. In the **Add Device** dialog box, the new product is automatically specified in the drop-down list of the Product section. Click Configure to add a custom feature to the product. For more information, see Add a TSL feature.



Create Self-Defined Feature

* Feature Type

Properties | Services | Events

* Feature Name

aaa

* Identifier

aaa

Unit

Select a unit

Description

Enter a description

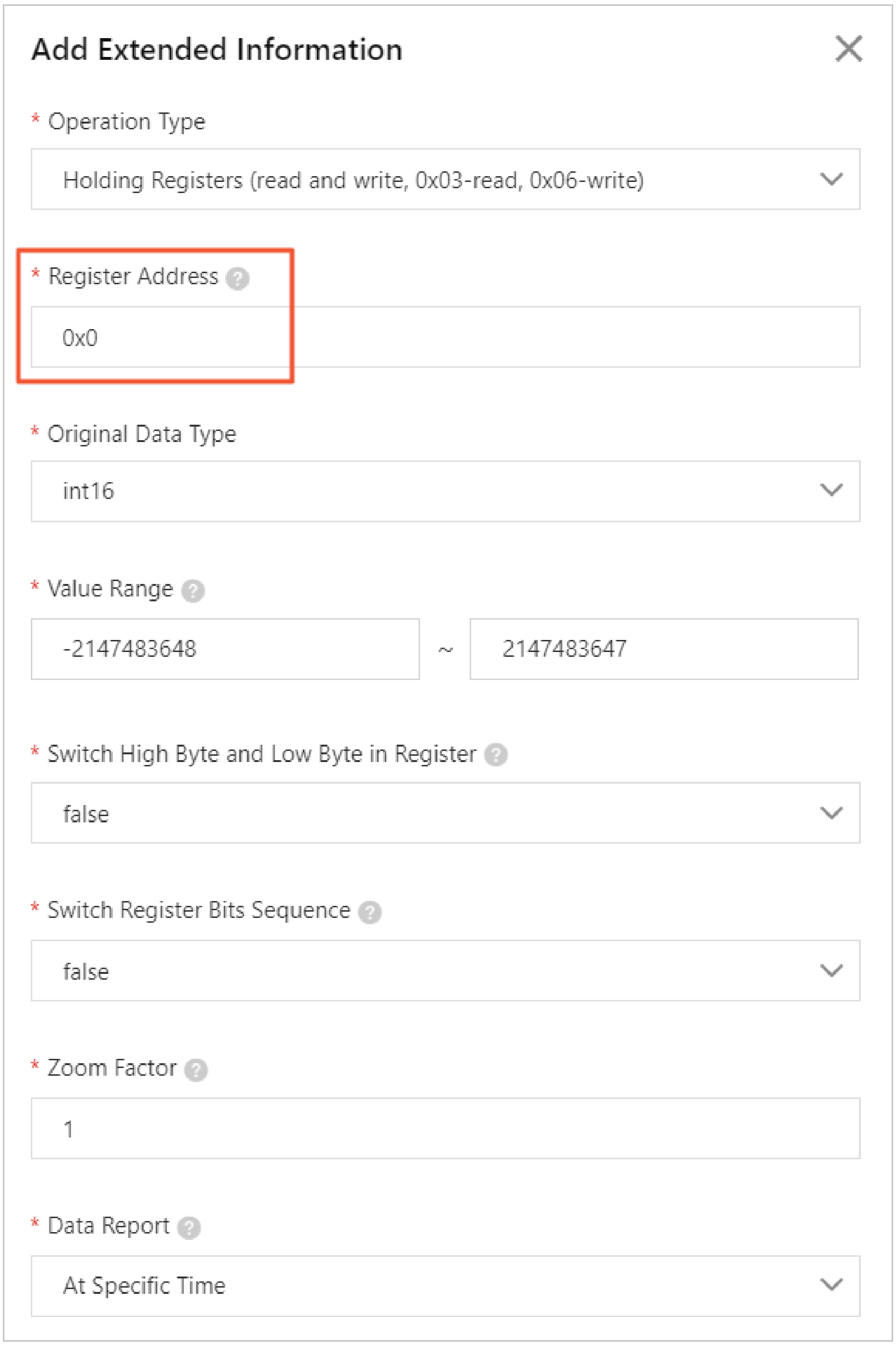
0/100

* Extended Information

+Add Extended Information

6. In the **Add Self-defined Feature** dialog box, click Properties and set the required parameters. Then, click **Add Extended Information**.

   In the Add Extended Information dialog box, set the required parameters to specify the data points of the Modbus sub-device.



Add Extended Information

* Operation Type

Holding Registers (read and write, 0x03-read, 0x06-write)

* Register Address

0x0

* Original Data Type

int16

* Value Range

-2147483648 ~ 2147483647

* Switch High Byte and Low Byte in Register

false

* Switch Register Bits Sequence

false

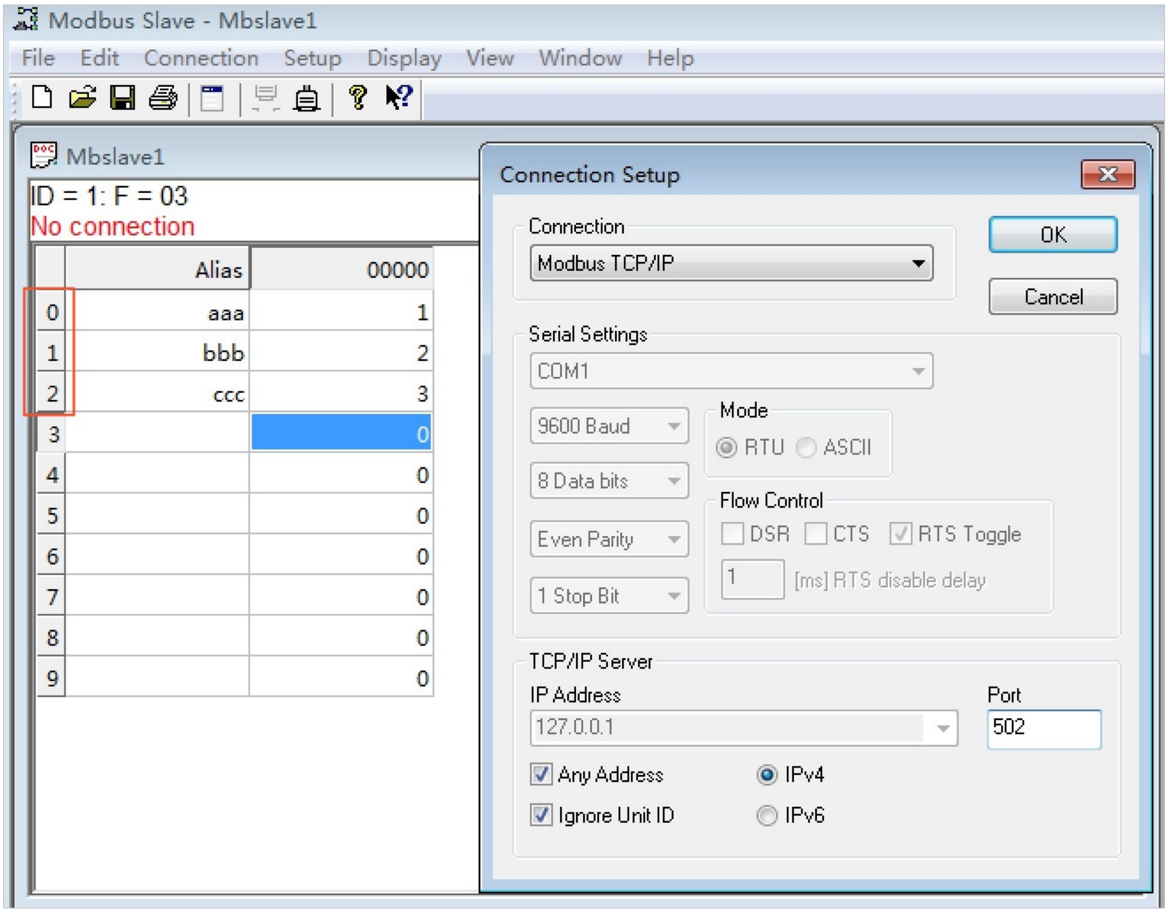
* Zoom Factor

1

* Data Report

At Specific Time

Set the Register Address parameter based on the data points of the simulated Modbus slave device, as shown in the following figure. In this example, three attributes named aaa, bbb, and ccc are created. These attributes correspond to the data points 0, 1, and 2 of the Modbus slave device.



Mbslave1

ID = 1: F = 03
No connection

	Alias	00000
0	aaa	1
1	bbb	2
2	ccc	3
3		0
4		0
5		0
6		0
7		0
8		0
9		0

Connection Setup

Connection: Modbus TCP/IP

Serial Settings: COM1

9600 Baud

8 Data bits

Even Parity

1 Stop Bit

Mode: ● RTU ○ ASCII

Flow Control: □ DSR □ CTS ☑ RTS Toggle

1 [ms] RTS disable delay

TCP/IP Server

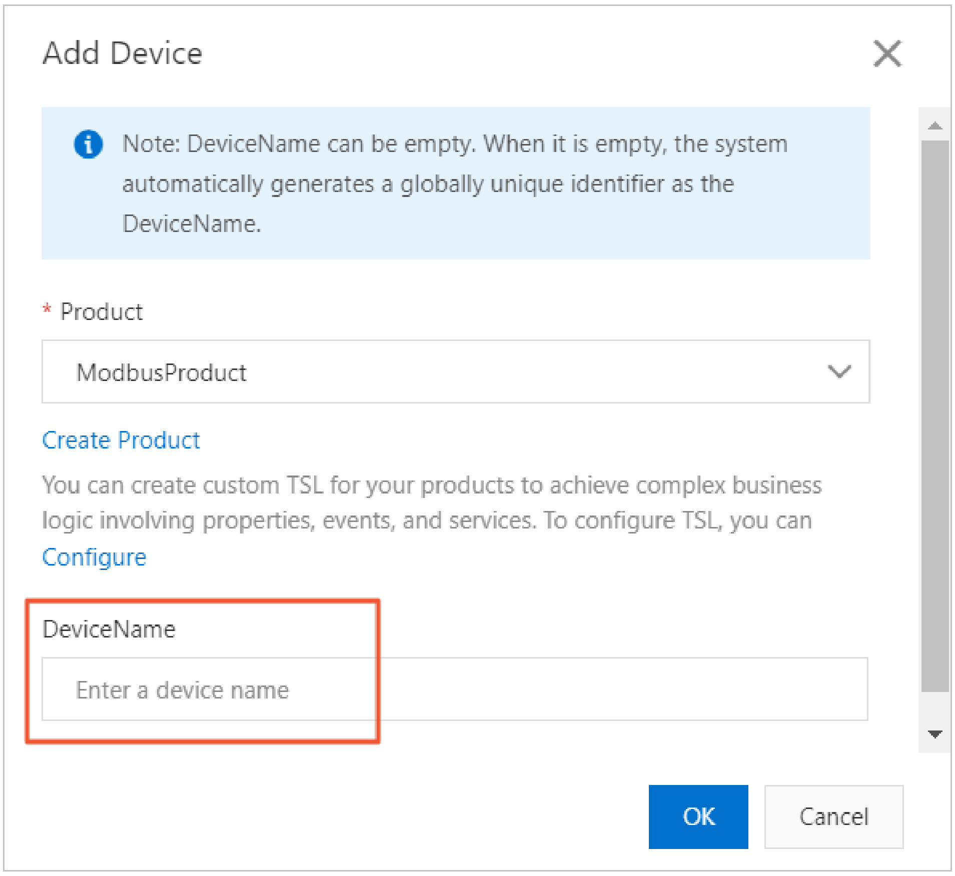
IP Address: 127.0.0.1

Port: 502

☑ Any Address ● IPv4

☑ Ignore Unit ID ○ IPv6

7. Go to the **Add Device** dialog of the **Instance Details** page in the Link IoT Edge console. Then, add a Modbus device.



Add Device

Note: DeviceName can be empty. When it is empty, the system automatically generates a globally unique identifier as the DeviceName.

* Product

ModbusProduct

Create Product

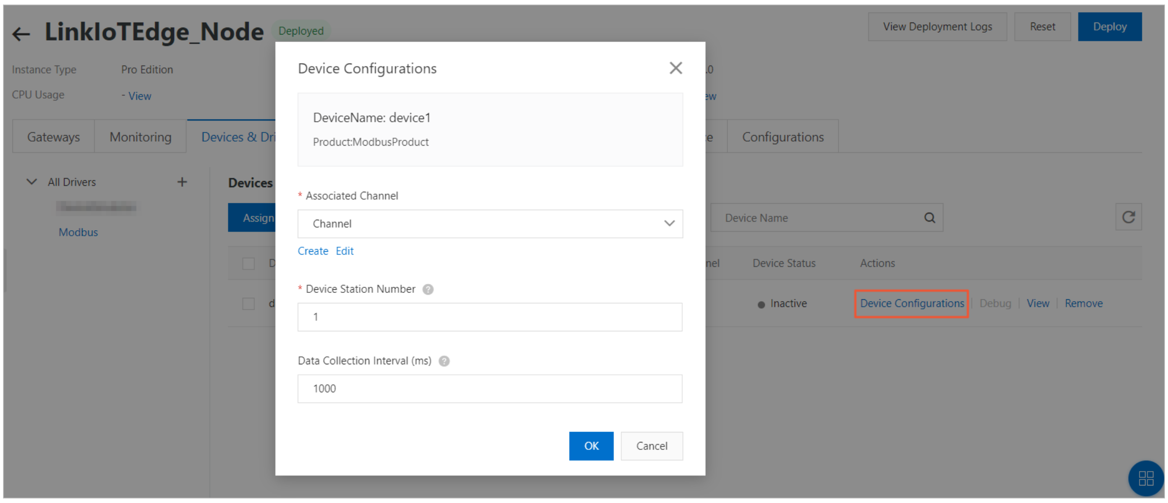
You can create custom TSL for your products to achieve complex business logic involving properties, events, and services. To configure TSL, you can Configure

DeviceName

Enter a device name

Step 3: Configure and deploy the edge instance

1. Assign the new Modbus device to the edge instance.
2. After the device is assigned to the edge instance, click **Device Configurations** in the Actions column of the device. Then, use a channel to associate the device with the Modbus driver.



Device Configurations

DeviceName: device1

Product:ModbusProduct

* Associated Channel: Channel

* Device Station Number: 1

Data Collection Interval (ms): 1000

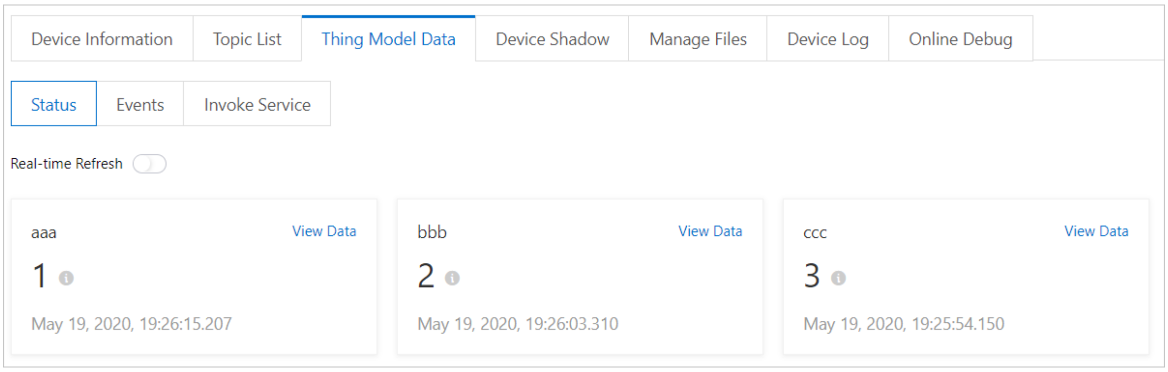
Parameters

Parameter	Description
Associated Channel	Select the channel that you have added in the Step 1: Assign a Modbus driver to the edge instance section.

Parameter	Description
Device Station Number	To view the value, check the ID of the Modbus slave device in Step 6 of the Step 2: Assign a sub-device to the Modbus driver section. In this example, the value of this parameter is 1.

3. On the **Instance Details** page, click **Deploy** in the upper-right corner of the page to deploy the edge instance.
4. Log on to the IoT Platform console. In the left-side navigation pane, choose **Devices > Devices**. On the Devices page, click **View** next to the required Modbus product.

   On the **Device Details** page, choose **TSL Data > Status**. On the Status tab, view the properties of the Modbus product.



aaa	bbb	ccc
1	2	3
May 19, 2020, 19:26:15.207	May 19, 2020, 19:26:03.310	May 19, 2020, 19:25:54.150

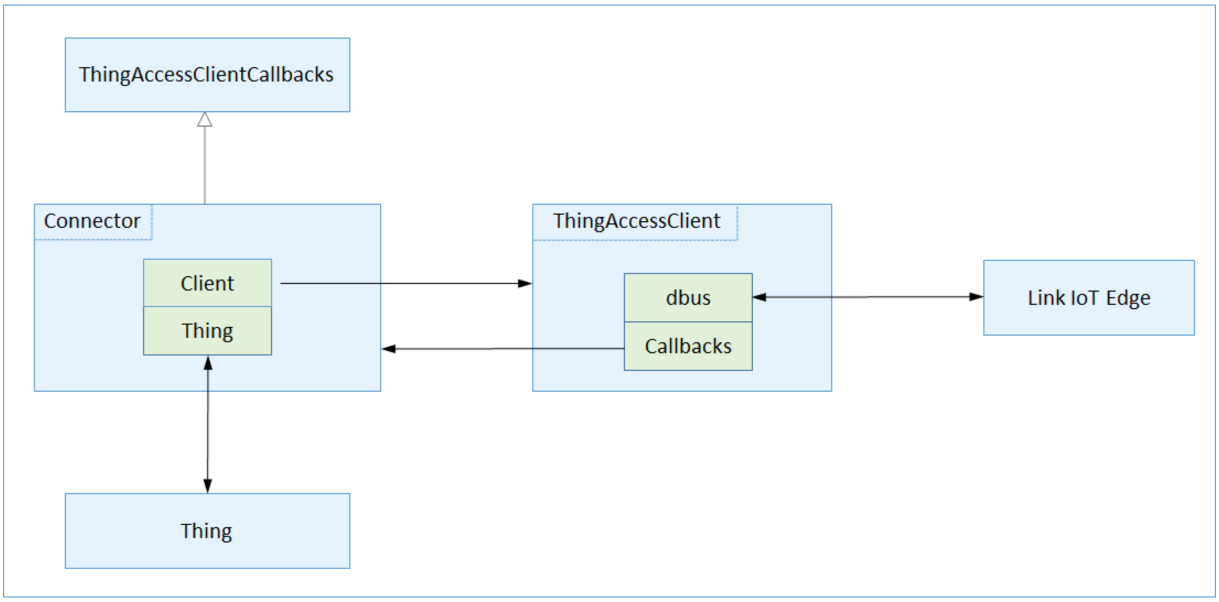
2.Develop a driver with the Connector architecture

This topic describes how to develop a driver with the Connector architecture, which is clear and flexible. For your convenience, we recommend that you use the Connector architecture to develop drivers.

Currently, the Connector architecture is only applicable to device SDKs developed in Node.js and Python.

Overview

The following figure shows the Connector architecture.



A driver with the Connector architecture consists of the following classes:

- ThingAccessClient

  The ThingAccessClient class is encapsulated in a device SDK and provides multiple methods for sub-devices to send data to and receive data from Link IoT Edge. The ThingAccessClient class can call callback functions of the ThingAccessClientCallbacks class. When receiving a request with the pointer of a callback function specified, the ThingAccessClient class obtains required data from Link IoT Edge and then calls the callback function. In the Connector architecture, callback functions of the ThingAccessClientCallbacks class are implemented in the Connector class.

- Connector

  The Connector class is the core of the Connector architecture. It provides the connect method for connecting sub-devices to Link IoT Edge and the disconnect method for disconnecting sub-devices from Link IoT Edge. In addition, the Connector class supports interfaces encapsulated by the Thing class for sub-devices to connect to Link IoT Edge. The Connector class implements callback functions of the ThingAccessClientCallbacks class. When constructing a ThingAccessClient object, the Connector class specifies the pointer of a callback function and transmits the pointer to the ThingAccessClient class. When receiving required data from Link IoT Edge, the ThingAccessClient class calls the callback function.

- Thing

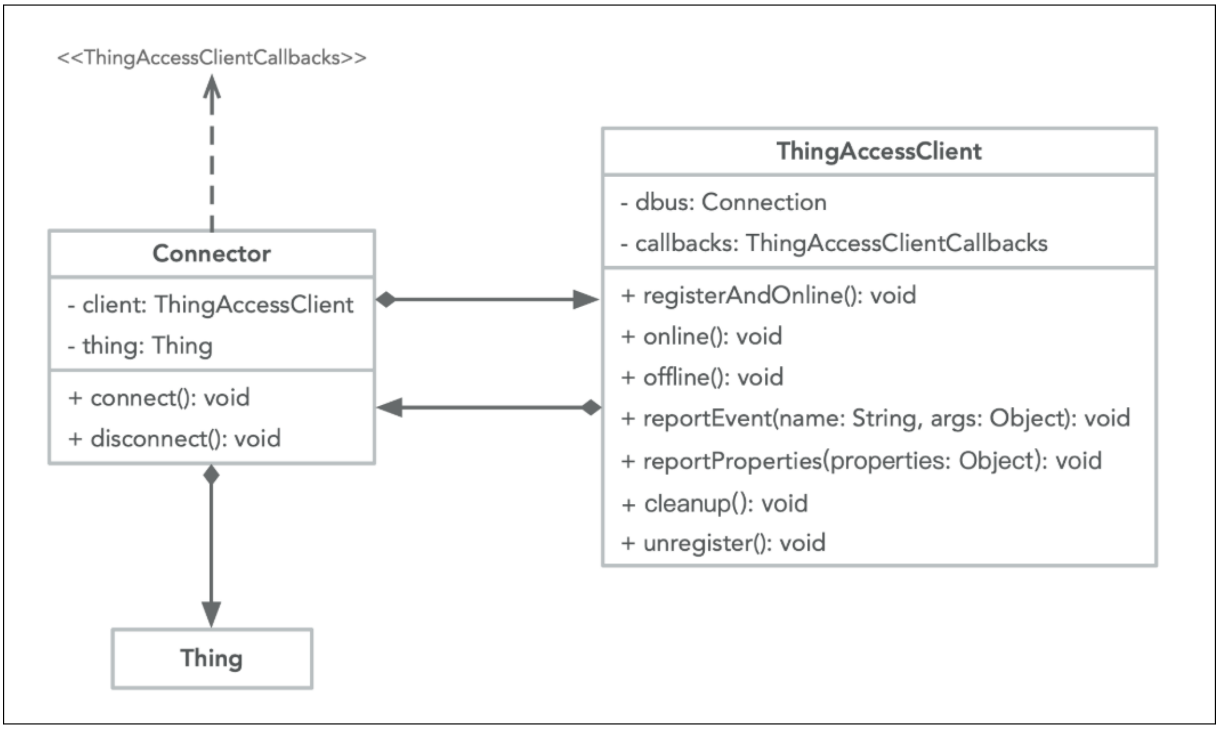
The Thing class directly interacts with sub-devices. It encapsulates interfaces of physical sub-devices for the Connector class to call, and provides object-oriented API operations for sub-devices to call. When the driver connects to a specific sub-device, the Thing class refers to the abstract class of the sub-device, for example, the Light class of a light.

- Entry

  The Entry class is the main entry point of a driver. It obtains the driver configuration, initializes the Thing and Connector classes, and then calls the connect method to connect a sub-device to Link IoT Edge. The Entry class can also call the disconnect method to disconnect a sub-device from Link IoT Edge.

The Connector class connects the abstract class of a sub-device and that of Link IoT Edge by combining the classes, hence the name. In this example, the abstract class of a sub-device and that of Link IoT Edge are Thing and ThingAccessClient, respectively.

The following figure shows a Unified Modeling Language (UML) class diagram.


<<ThingAccessClientCallbacks>>
Connector
- client: ThingAccessClient
- thing: Thing
+ connect(): void
+ disconnect(): void
ThingAccessClient
- dbus: Connection
- callbacks: ThingAccessClientCallbacks
+ registerAndOnline(): void
+ online(): void
+ offline(): void
+ reportEvent(name: String, args: Object): void
+ reportProperties(properties: Object): void
+ cleanup(): void
+ unregister(): void
Thing

Procedure

This section describes how to use a Node.js SDK to develop a driver with the Connector architecture. For more information about how to use a Python SDK to develop a driver, see Link IoT Edge Thing Access SDK for Python.

Light

To develop a driver for a simulated light, follow these steps:

1. Define an abstract class for the simulated light that can be turned on or off by changing the value of the isOn property to *true* or *false*.

   The sample code is as follows:

```
/**
 * Define an abstract class for the simulated light that can be turned on or off by cha
nging the value of the
 * <code>isOn</code> property.
 */
class Light {
  constructor() {
    this._isOn = true;
  }
  get isOn() {
    return this._isOn;
  }
  set isOn(value) {
    return this._isOn = value;
  }
```

2. Define the Connector class. The Connector class provides the following features:
    - Receives the configuration and an abstract object of the simulated light and constructs a ThingAccessClient object for interacting with Link IoT Edge.
    - Implements three callback functions of the ThingAccessClientCallbacks class and uses the callback functions to call interfaces encapsulated by the Light class.
    - Provides the connect and disconnect methods. The connect method can connect the simulated light to Link IoT Edge and the disconnect method can disconnect the simulated light from Link IoT Edge.

    The sample code is as follows:

```
/**
 * Construct a class to combine ThingAccessClient and the abstract class of the simulat
ed light that connects
 * to Link IoT Edge.
 */
class Connector {
  constructor(config, light) {
    this.config = config;
    this.light = light;
    this._client = new ThingAccessClient(config, {
      setProperties: this._setProperties.bind(this),
      getProperties: this._getProperties.bind(this),
      callService: this._callService.bind(this),
    });
  }
  /**
   * Connect to Link IoT Edge and publish properties to it.
   */
  connect() {
    registerAndOnlineWithBackOffRetry(this._client, 1)
      .then(() => {
        return new Promise(() => {
          // Publish properties to Link IoT Edge.
          const properties = { 'LightSwitch': this.light.isOn ? 1 : 0 };
          this._client.reportProperties(properties);
        });
      })
      .catch(err => {
        console.log(err);
        return this._client.cleanup();
      })
      .catch(err => {
        console.log(err);
      });
  }
  /**
   * Disconnect from Link IoT Edge and stop publishing properties to it.
   */
  disconnect() {
    this._client.cleanup()
      .catch(err => {
        console.log(err);
      });
  }
  _setProperties(properties) {
    console.log('Set properties %s to thing %s-%s', JSON.stringify(properties),
      this.config.productKey, this.config.deviceName);
    if ('LightSwitch' in properties) {
      var value = properties['LightSwitch'];
      var isOn = value === 1;
      if (this.light.isOn !== isOn) {
        // Report property changes to Link IoT Edge.
        this.light.isOn = isOn;
        if (this._client) {
          properties = {'LightSwitch': value};
          console.log(`Report properties: ${JSON.stringify(properties)}`);
          this._client.reportProperties(properties);
        }
      }
      return {
        code: RESULT_SUCCESS,
        message: 'success',
      };
    }
    return {
      code: RESULT_FAILURE,
      message: 'The requested properties does not exist.',
    };
  }
  _getProperties(keys) {
    console.log('Get properties %s from thing %s-%s', JSON.stringify(keys),
      this.config.productKey, this.config.deviceName);
    if (keys.includes('LightSwitch')) {
      return {
        code: RESULT_SUCCESS,
        message: 'success',
        params: {
          'LightSwitch': this.light.isOn ? 1 : 0,
        }
      };
    }
    return {
      code: RESULT_FAILURE,
      message: 'The requested properties does not exist.',
    }
  }
  _callService(name, args) {
    console.log('Call service %s with %s on thing %s-%s', JSON.stringify(name),
      JSON.stringify(args), this.config.productKey, this.config.deviceName);
    return {
      code: RESULT_FAILURE,
      message: 'The requested service does not exist.',
    };
  }
}
```

3. Obtain the driver configuration and initialize the Connector class.
   - Call the getConfig operation to obtain the driver configuration.
   - Call the getThingInfos operation to obtain the information about and configuration of the simulated light.
   - Initialize the Connector class.
   - Call the connect method to connect the simulated light to Link IoT Edge.

   The sample code is as follows:

```
// Obtain the configuration that is automatically generated when the simulated light is
bound to this driver. getConfig()
  .then((config) => {
    // Obtain the simulated light information, for example, the product key and device
    // name of the simulated light, from config.
    const thingInfos = config.getThingInfos();
    thingInfos.forEach((thingInfo) => {
      const light = new Light();
      // The value format of the ThingInfo parameter is supported by config of Connecto
r. Pass the ThingInfo parameter directly.
      const connector = new Connector(thingInfo, light);
      connector.connect();
    });
  });
```

LightSensor

To develop a driver for a simulated light sensor, follow these steps:

1. Define an abstract class for the simulated light sensor that automatically runs when a listener listens to it and stops running when the listener is cleaned.

   The sample code is as follows:

```
/**
 * Define an abstract class for the simulated light sensor that starts to publish illum
inance between 100
 * and 600 with 100 delta changes when a listener listens to it.
 */
class LightSensor {
  constructor() {
    this._illuminance = 200;
    this._delta = 100;
  }
  get illuminance() {
    return this._illuminance;
  }
  // Start to work.
  start() {
    if (this._clearInterval) {
      this._clearInterval();
    }
    console.log('Starting light sensor...');
    const timeout = setInterval(() => {
      // Update illuminance and delta.
      let delta = this._delta;
      let illuminance = this._illuminance;
      if (illuminance >= 600 || illuminance <= 100) {
        delta = -delta;
      }
      illuminance += delta;
      this._delta = delta;
      this._illuminance = illuminance;
      if (this._listener) {
        this._listener({
          properties: {
            illuminance,
          }
        });
      }
    }, 2000);
    this._clearInterval = () => {
      clearInterval(timeout);
      this._clearInterval = undefined;
    };
    return this._clearInterval;
  }
  stop() {
    console.log('Stopping light sensor ...');
    if (this._clearInterval) {
      this._clearInterval();
    }
  }
  listen(callback) {
    if (callback) {
      this._listener = callback;
      // Start to work when a listener listens to it.
      this.start();
    } else {
      this._listener = undefined;
      this.stop();
    }
  }
}
```

2. Define the Connector class. The Connector class provides the following features:
   - Receives the configuration and an abstract object of the simulated light sensor and constructs a ThingAccessClient object for interacting with Link IoT Edge.
   - Implements three callback functions of the ThingAccessClientCallbacks class and uses the callback functions to call interfaces encapsulated by the LightSensor class.
   - Provides the connect and disconnect methods. The connect method can connect the simulated light sensor to Link IoT Edge and the disconnect method can disconnect the simulated light sensor from Link IoT Edge.

   The sample code is as follows:

```
/**
 * Construct a class to combine ThingAccessClient and the abstract class of the simulat
ed light sensor that connects to Link IoT Edge. */
class Connector {
  constructor(config, lightSensor) {
    this.config = config;
    this.lightSensor = lightSensor;
    this._client = new ThingAccessClient(config, {
      setProperties: this._setProperties.bind(this),
      getProperties: this._getProperties.bind(this),
      callService: this._callService.bind(this),
    });
  }
  /**
   * Connect to Link IoT Edge and publish properties to it.   */
  connect() {
    registerAndOnlineWithBackOffRetry(this._client, 1)
      .then(() => {
        return new Promise(() => {
          // Run, listen to the simulated light sensor, and report property data change
s of the sensor to Link IoT Edge.
          this.lightSensor.listen((data) => {
            const properties = {'MeasuredIlluminance': data.properties.illuminance};
            console.log(`Report properties: ${JSON.stringify(properties)}`);
            this._client.reportProperties(properties);
          });
        });
      })
      .catch(err => {
        console.log(err);
        return this._client.cleanup();
      })
      .catch(err => {
        console.log(err);
      });
  }
  /**
   * Disconnect from Link IoT Edge.
   */
  disconnect() {
    // Clean the listener.
    this.lightSensor.listen(undefined);
    this._client.cleanup()
      .catch(err => {
        console.log(err);
      });
  }
  _setProperties(properties) {
    console.log('Set properties %s to thing %s-%s', JSON.stringify(properties),
      this.config.productKey, this.config.deviceName);
    return {
      code: RESULT_FAILURE,
      message: 'The property is read-only.',
    };
  }
  _getProperties(keys) {
    console.log('Get properties %s from thing %s-%s', JSON.stringify(keys),
      this.config.productKey, this.config.deviceName);
    if (keys.includes('MeasuredIlluminance')) {
      return {
        code: RESULT_SUCCESS,
        message: 'success',
        params: {
          'MeasuredIlluminance': this.lightSensor.illuminance,
        }
      };
    }
    return {
      code: RESULT_FAILURE,
      message: 'The requested properties does not exist.',
    }
  }
  _callService(name, args) {
    console.log('Call service %s with %s on thing %s-%s', JSON.stringify(name),
      JSON.stringify(args), this.config.productKey, this.config.deviceName);
    return {
      code: RESULT_FAILURE,
      message: 'The requested service does not exist.',
    };
  }
}
```

3. Obtain the driver configuration and initialize the Connector class.
   - Call the getConfig operation to obtain the driver configuration.
   - Call the getThingInfos operation to obtain the information about and configuration of the simulated light sensor.
   - Initialize the Connector class.
   - Call the connect method to connect the simulated light sensor to Link IoT Edge.

   The sample code is as follows:

```
// Obtain the configuration that is automatically generated when the simulated light se
nsor is bound to this driver. getConfig()
  .then((config) => {
    // Obtain the information about the simulated light sensor, for example, the produc
t key and device    // name of the simulated light sensor, from config.    const thingI
nfos = config.getThingInfos();
    thingInfos.forEach((thingInfo) => {
      const lightSensor = new LightSensor();
      // The value format of the ThingInfo parameter is supported by config of Connecto
r. Pass the ThingInfo parameter directly.      const connector = new Connector(thingInf
o, lightSensor);
      connector.connect();
    });
  });
```

3.Connect an OPC UA sub-device to a gateway

This topic describes how to connect an OPC Unified Architecture (OPC UA) sub-device to a gateway and enable the sub-device to interact with IoT Platform.

Prerequisites

- A Docker runtime environment is built for Link IoT Edge Pro.
- An edge instance is created and the gateway assigned to the edge instance is brought online. For more information, see Link IoT Edge Pro.

Step 1: Build an OPC UA Server

The following table describes the environment requirements for an OPC UA Server.

Item	Version	Installation command
Python	3.5.2 or later	None
PIP	9.0.1 or later	None
OPC UA	0.98.3 or later	**pip install opcua==0.98.3**

To build an OPC UA Server to simulate an LED light that is named demo_led and has the temperature property and high_temperature event, follow these steps:

1. Run the following command to download the package of the OPC UA Server:

```
wget http://iotedge-web.oss-cn-shanghai.aliyuncs.com/public/driverSample/opcua_simulation_server.tar.gz
```

2. Run the following commands to start the OPC UA Server:

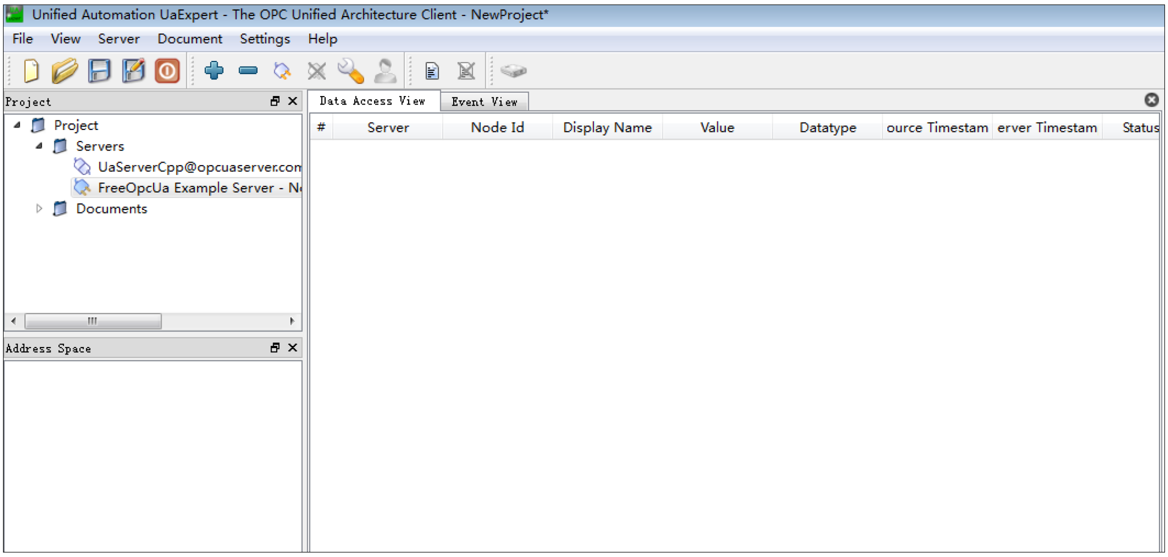
```
tar -zxvf opcua_simulation_server.tar.gz
cd opcua_simulation_server && ./opcua_simulation_server.sh
```

Step 2: Install an OPC UA client

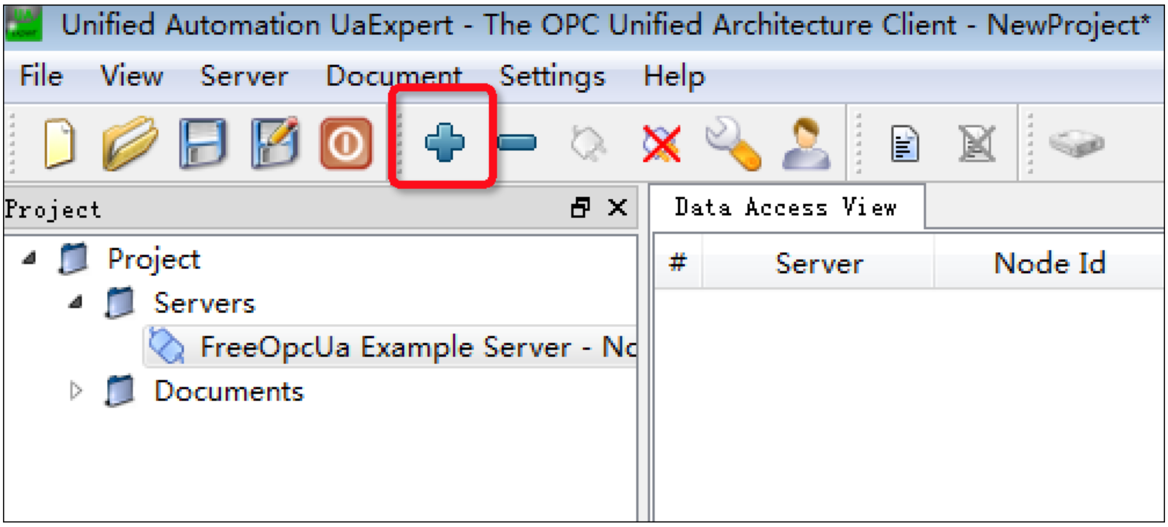
Before connecting the simulated LED light to a gateway through an OPC UA driver, you must configure the simulated LED light. When configuring the simulated LED light, you must use an OPC UA client to obtain the information about the simulated LED light from the OPC UA Server. The obtained information is required when you create a product and configure the driver in the IoT Platform console.

In this example, UaExpert is used as an OPC UA client.

1. Download and install UaExpert.
2. Start UaExpert.

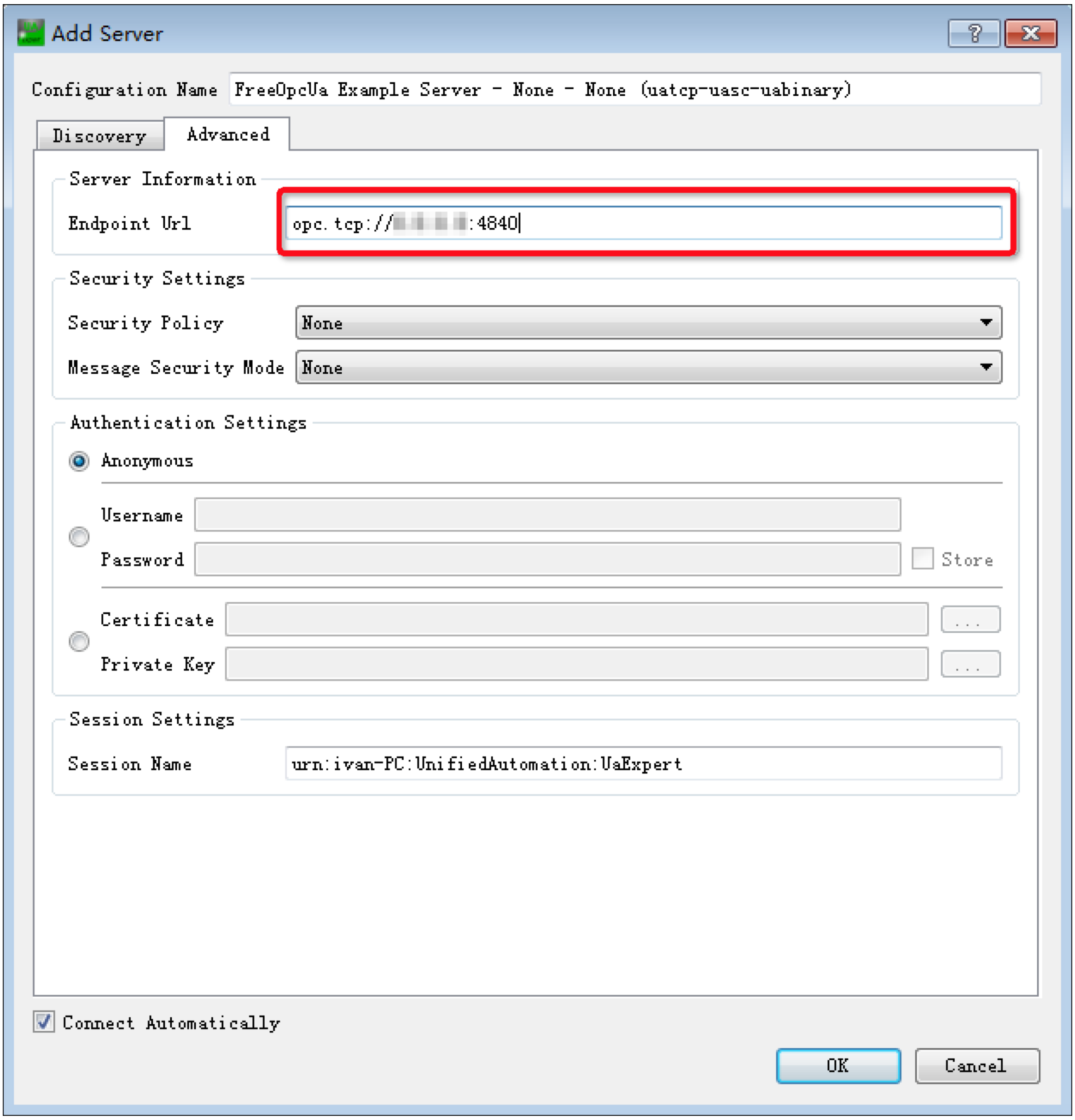

Unified Automation UaExpert - The OPC Unified Architecture Client - NewProject*
File View Server Document Settings Help
Project
Project
Servers
UaServerCpp@opcuaserver.com
FreeOpcUa Example Server - N
Documents
Address Space
Data Access View
Event View
# Server Node Id Display Name Value Datatype ource Timestam erver Timestam Status

3. Click + in the toolbar.


Unified Automation UaExpert - The OPC Unified Architecture Client - NewProject*
File View Server Document Settings Help
Project
Project
Servers
FreeOpcUa Example Server - Nc
Documents
Data Access View
# Server Node Id

4. In the **Add Server** dialog box that appears, click the **Advanced** tab and set Endpoint Url to the URL of the OPC UA Server. In the URL, specify the IP address and port number of the host where the OPC UA Server resides in the `Host IP address:Port number` format.

> **Note** For example, if the default port number of the OPC UA Server is `4840` and the IP address is 192.168.1.1, set the Endpoint Url parameter to the following value:
>
> ```
> opc.tcp://192.168.1.1:4840
> ```



Configuration Name: FreeOpcUa Example Server - None - None (uatcp-uasc-uabinary)

Endpoint Url: opc.tcp://[redacted]:4840

Security Policy: None

Message Security Mode: None

● Anonymous

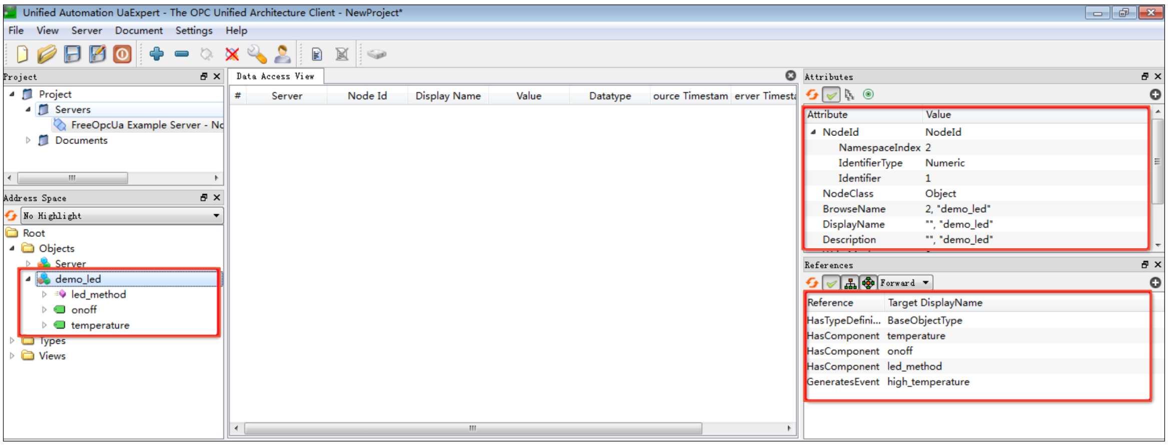
○ Username:
Password: □ Store

○ Certificate: ...
Private Key: ...

Session Name: urn:ivan-PC:UnifiedAutomation:UaExpert

☑ Connect Automatically

5. Click **OK**. The information about the simulated LED light appears.



```
Project
  Servers
    FreeOpcUa Example Server - Nc
  Documents
```

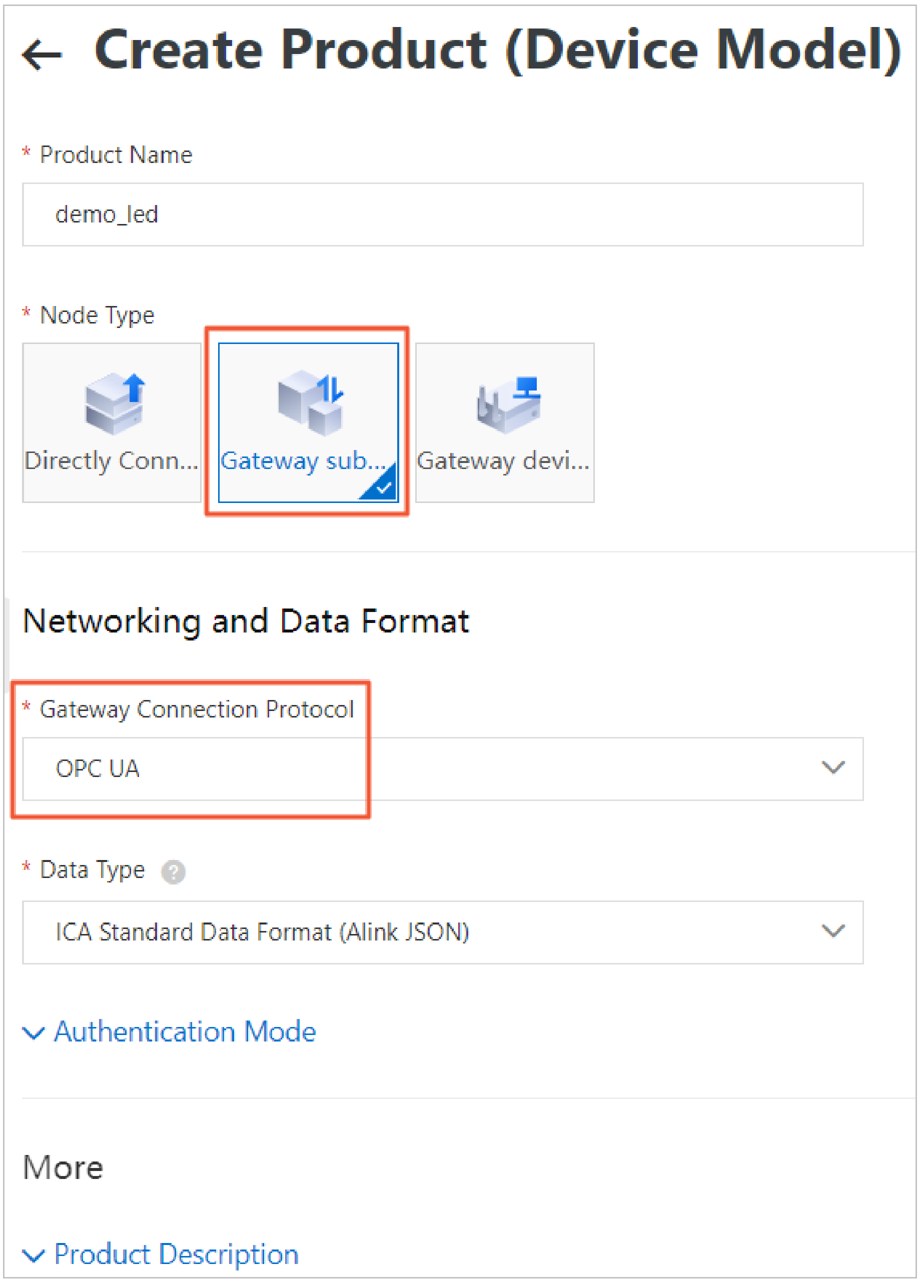
```
Root
  Objects
    Server
    demo_led
      led_method
      onoff
      temperature
  Types
  Views
```

Attribute	Value
NodeId	NodeId
NamespaceIndex	2
IdentifierType	Numeric
Identifier	1
NodeClass	Object
BrowseName	2, "demo_led"
DisplayName	"", "demo_led"
Description	"", "demo_led"

Reference	Target DisplayName
HasTypeDefini...	BaseObjectType
HasComponent	temperature
HasComponent	onoff
HasComponent	led_method
GeneratesEvent	high_temperature

Step 3: Add the simulated LED light as an OPC UA sub-device

1. Create an OPC UA product. For more information, see Create a product.



← Create Product (Device Model)

* Product Name

demo_led

* Node Type

Networking and Data Format

* Gateway Connection Protocol

OPC UA

* Data Type

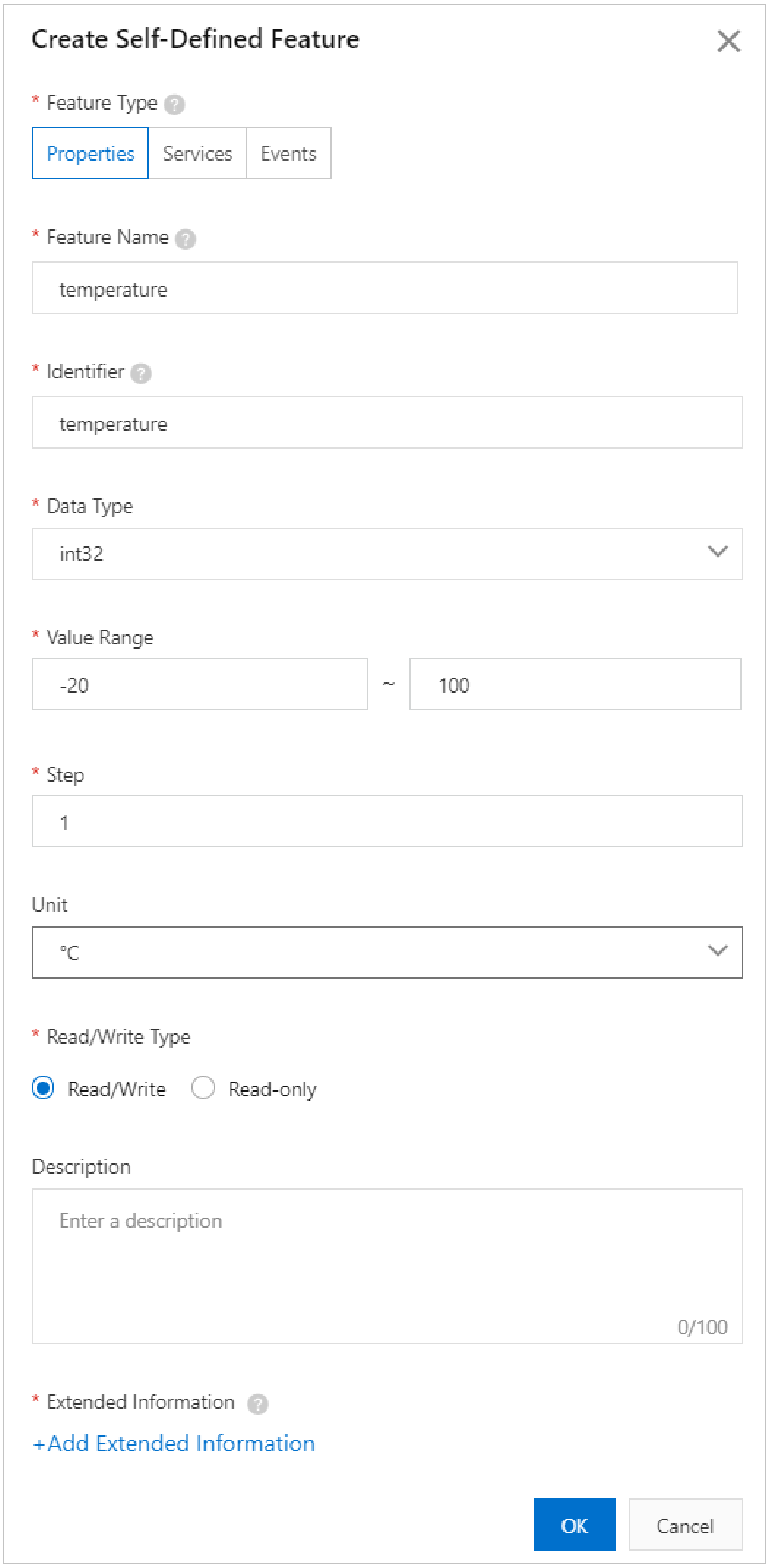
ICA Standard Data Format (Alink JSON)

More

The following table describes some required parameters.

Parameter	Description
Node Type	Select **Gateway sub-device**.
Gateway Connection Protocol	Select **OPC UA**.

2. Add custom features for the product. For more information, see Add a TSL feature.
   - *Add a property*
     a. Set required parameters for the property, as shown in the following figure.



Create Self-Defined Feature

* Feature Type

* Feature Name

temperature

* Identifier

temperature

* Data Type

int32

* Value Range

-20 ~ 100

* Step

1

Unit

°C

* Read/Write Type

● Read/Write ○ Read-only

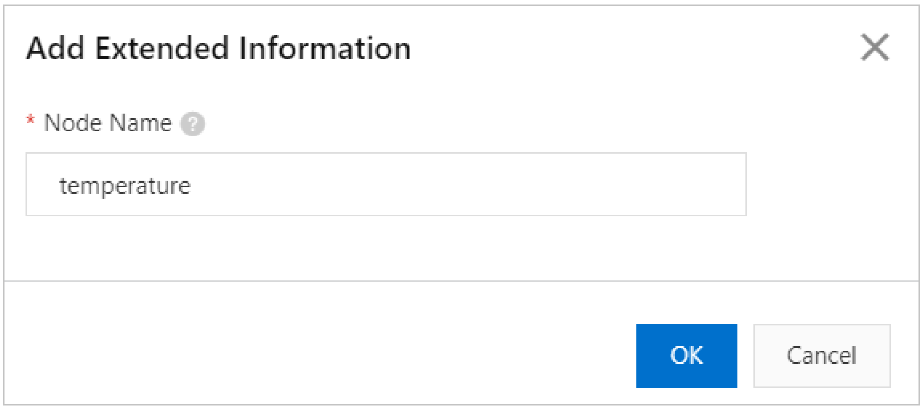
Description

Enter a description

0/100

* Extended Information

b. Click **Add Extended Information**. In the **Add Extended Information** dialog box that appears, set Node Name.

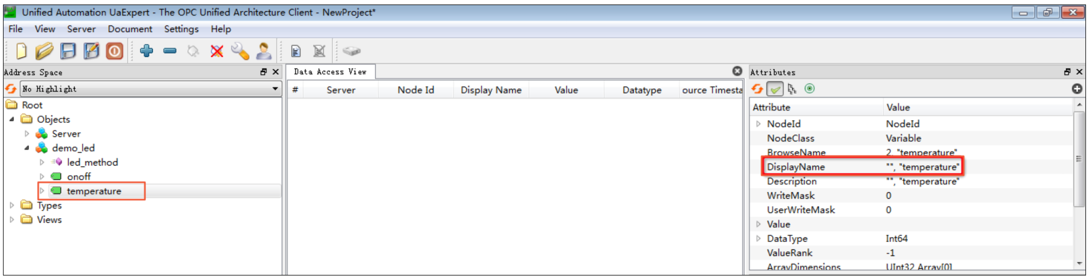


Add Extended Information

* Node Name

temperature

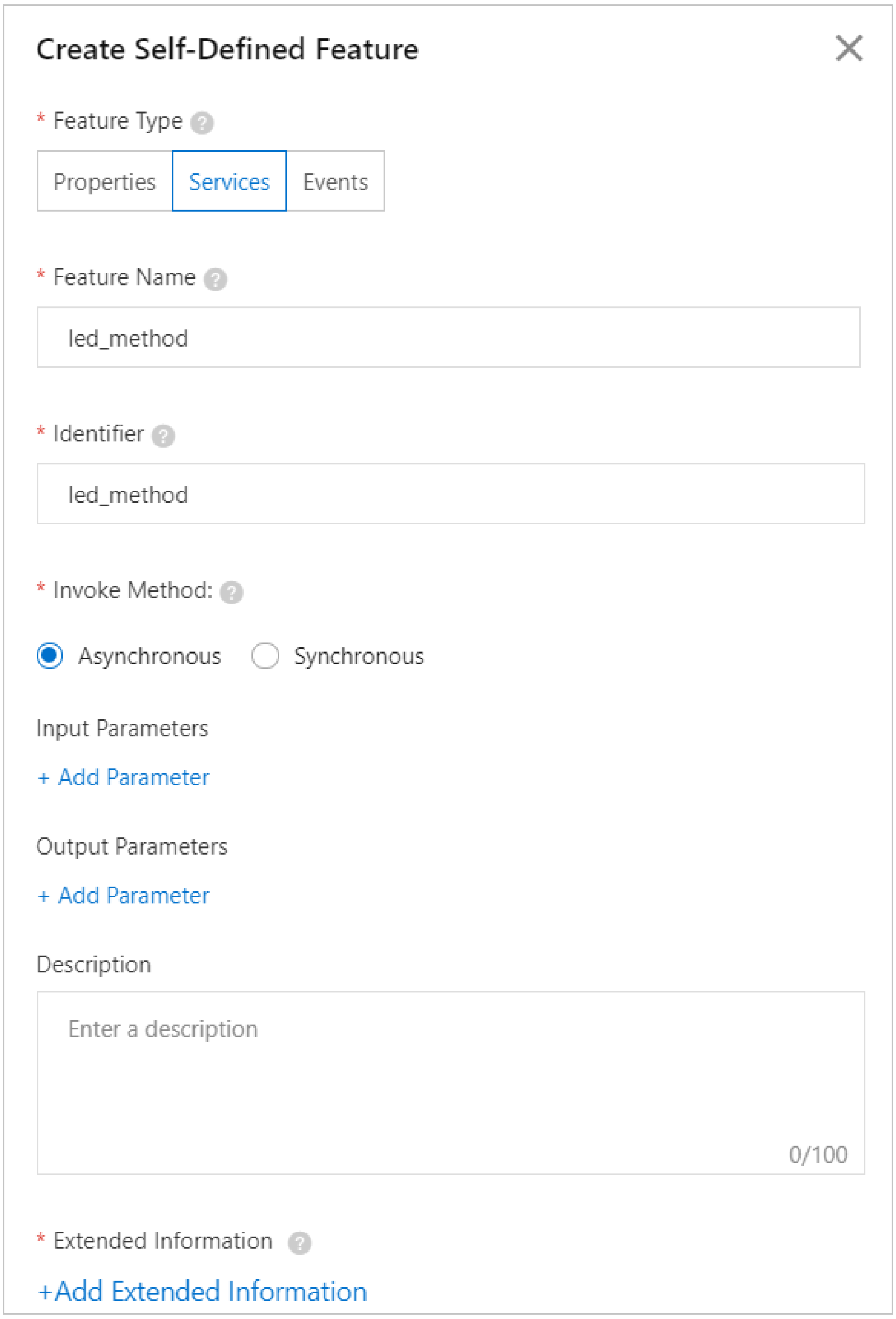
To obtain the value of Node Name, find the simulated LED light named demo_led in UaExpert and click temperature under demo_led. Check the value of DisplayName in the **Attributes** section on the right and use it as the value of Node Name.



Attribute	Value
NodeId	NodeId
NodeClass	Variable
BrowseName	2, "temperature"
DisplayName	"", "temperature"
Description	"", "temperature"
WriteMask	0
UserWriteMask	0
Value	
DataType	Int64
ValueRank	-1
ArrayDimensions	UInt32 Array[0]

- *Add a service*

a. Set required parameters for the service, as shown in the following figure.



Create Self-Defined Feature

* Feature Type

Properties | Services | Events

* Feature Name

led_method

* Identifier

led_method

* Invoke Method:

● Asynchronous ○ Synchronous

Input Parameters

+ Add Parameter

Output Parameters

+ Add Parameter

Description

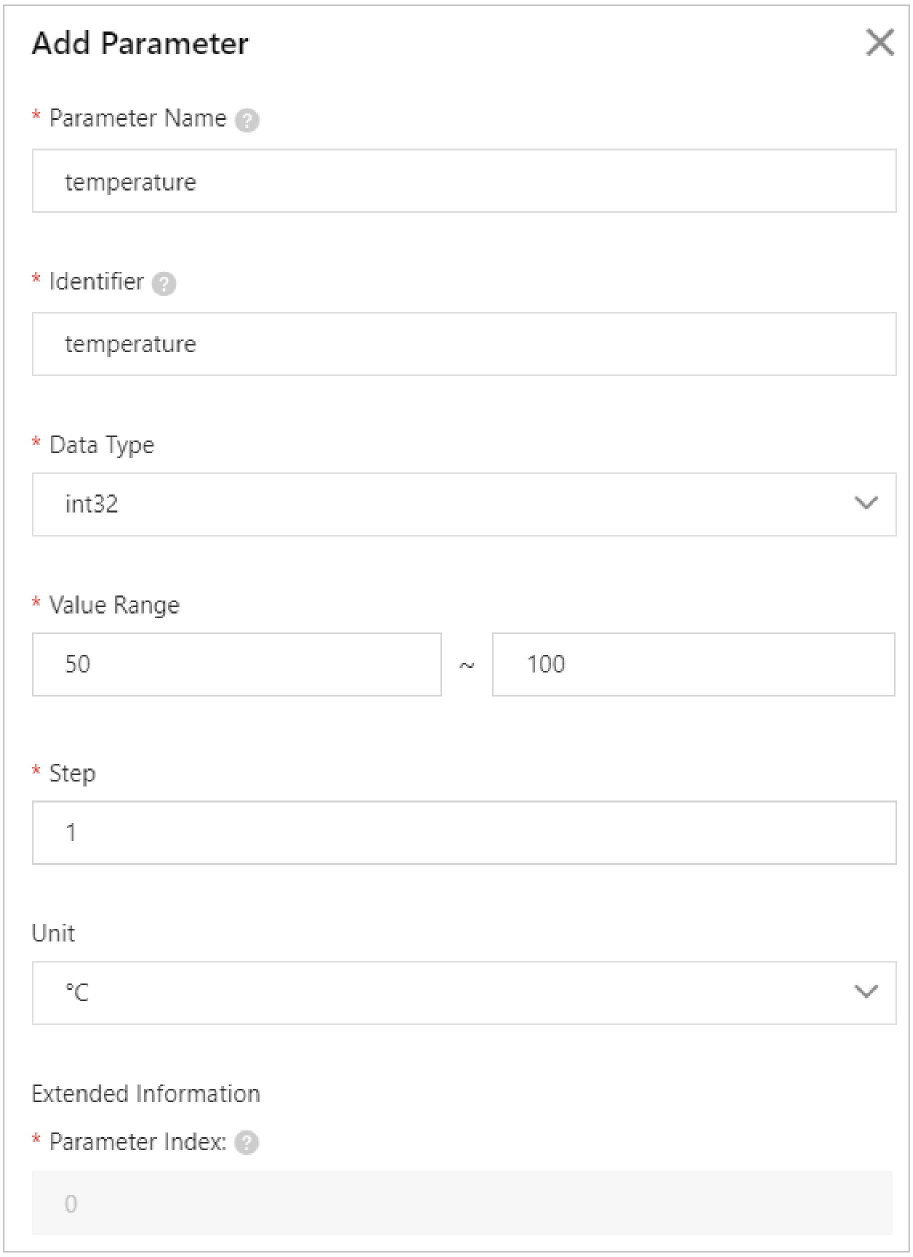
Enter a description

0/100

* Extended Information

+Add Extended Information

b. Click **Add Parameter** under Input Parameters to add a parameter for the service.



Add Parameter

* Parameter Name

temperature

* Identifier

temperature

* Data Type

int32

* Value Range

50 ~ 100

* Step

1

Unit

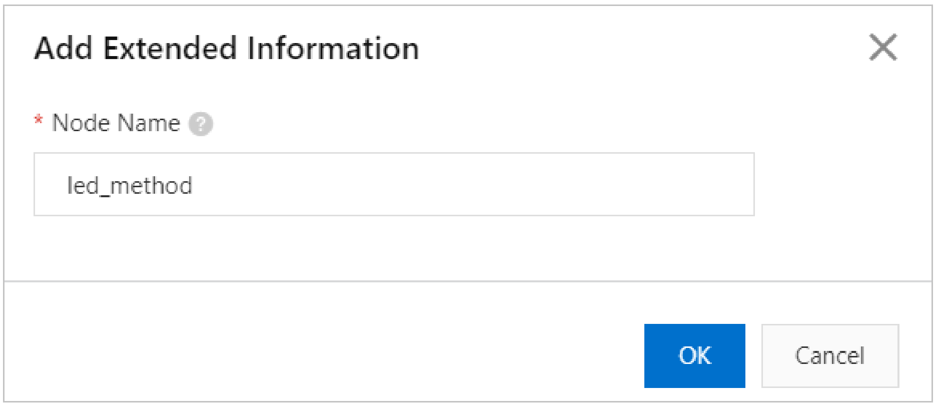
°C

Extended Information

* Parameter Index:

0

c. Click **Add Extended Information**. In the **Add Extended Information** dialog box that appears, set Node Name.

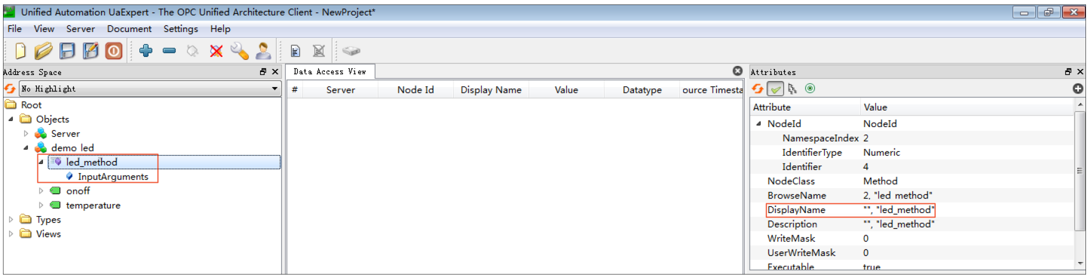


Add Extended Information

* Node Name

led_method

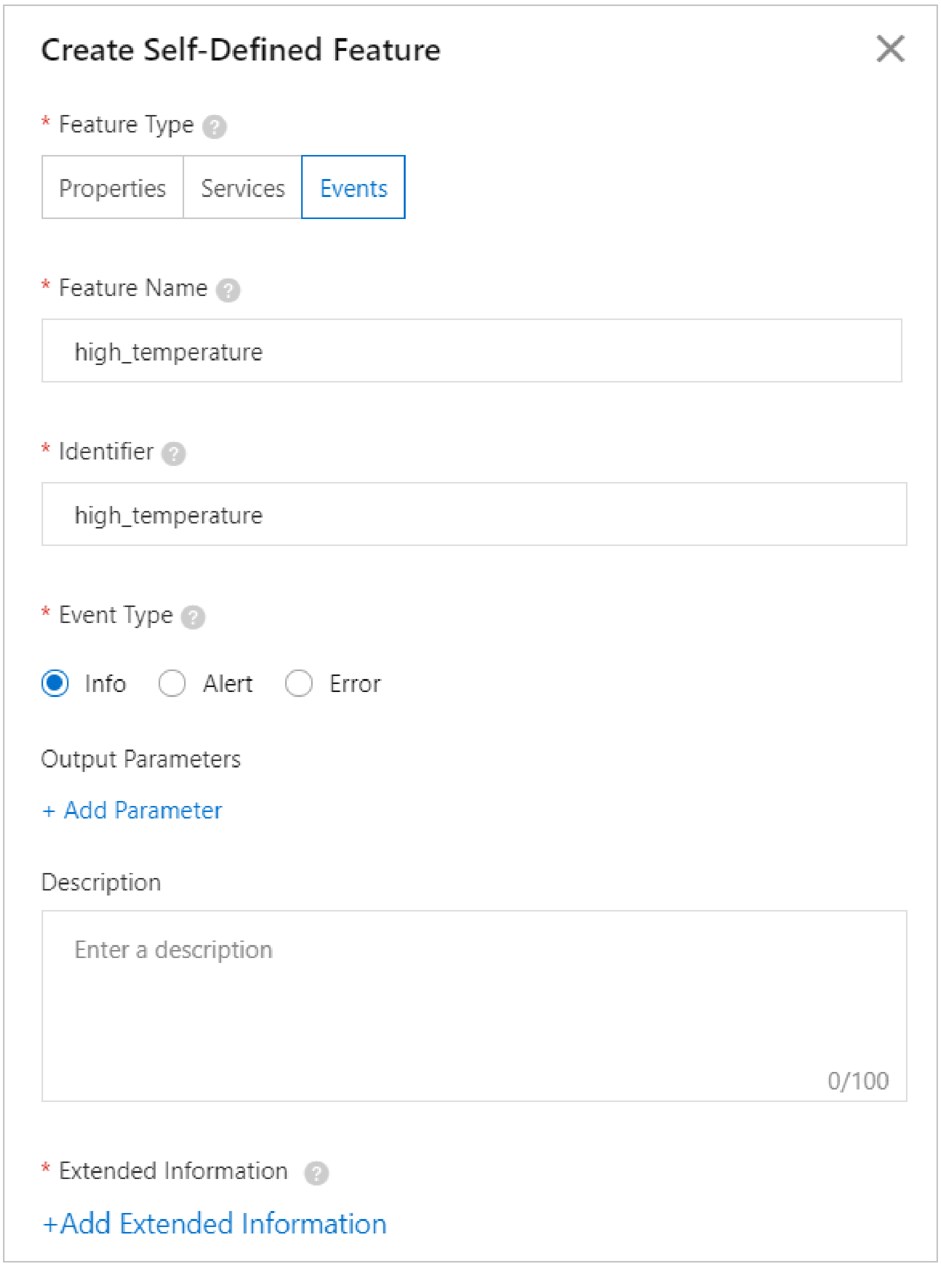
To obtain the value of Node Name, find the simulated LED light named demo_led in UaExpert and click led_method under demo_led. Check the value of DisplayName in the **Attributes** section on the right and use it as the value of Node Name.



Attribute	Value
NodeId	NodeId
NamespaceIndex	2
IdentifierType	Numeric
Identifier	4
NodeClass	Method
BrowseName	2, "led method"
DisplayName	"", "led_method"
Description	"", "led_method"
WriteMask	0
UserWriteMask	0
Executable	true

- *Add an event*

a. Set required parameters for the event, as shown in the following figure.



Create Self-Defined Feature

* Feature Type

Events

* Feature Name

high_temperature

* Identifier

high_temperature

* Event Type

● Info ○ Alert ○ Error

Output Parameters

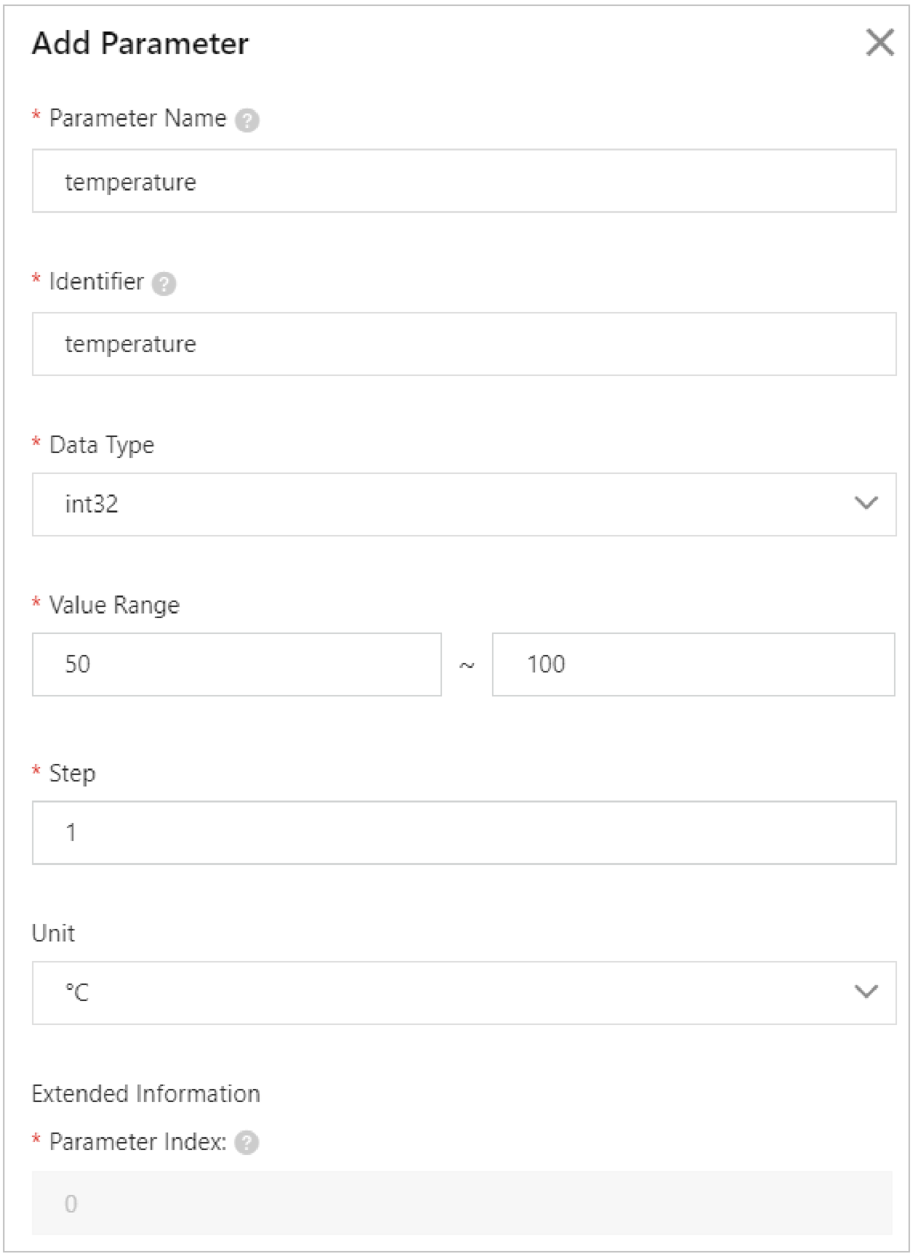
Description

Enter a description

0/100

* Extended Information

b. Click **Add Parameter** under Output Parameters to add a parameter for the event.



Add Parameter

* Parameter Name ?

temperature

* Identifier ?

temperature

* Data Type

int32

* Value Range

50 ~ 100

* Step

1

Unit

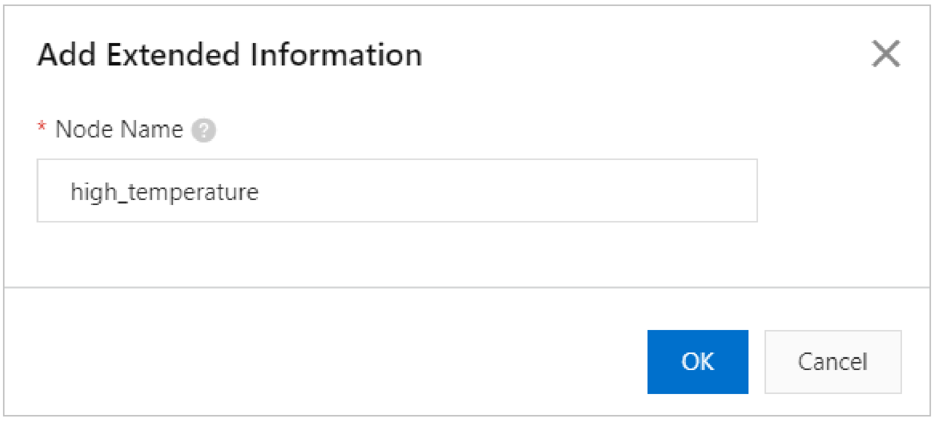
°C

Extended Information

* Parameter Index: ?

0

c. Click **Add Extended Information**. In the **Add Extended Information** dialog box that appears, set Node Name.

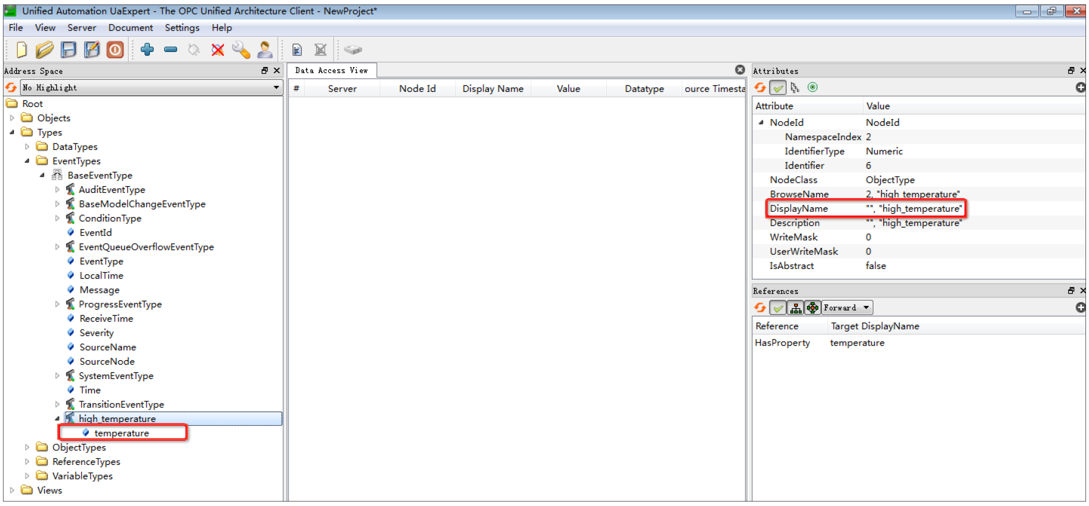


Add Extended Information

* Node Name

high_temperature

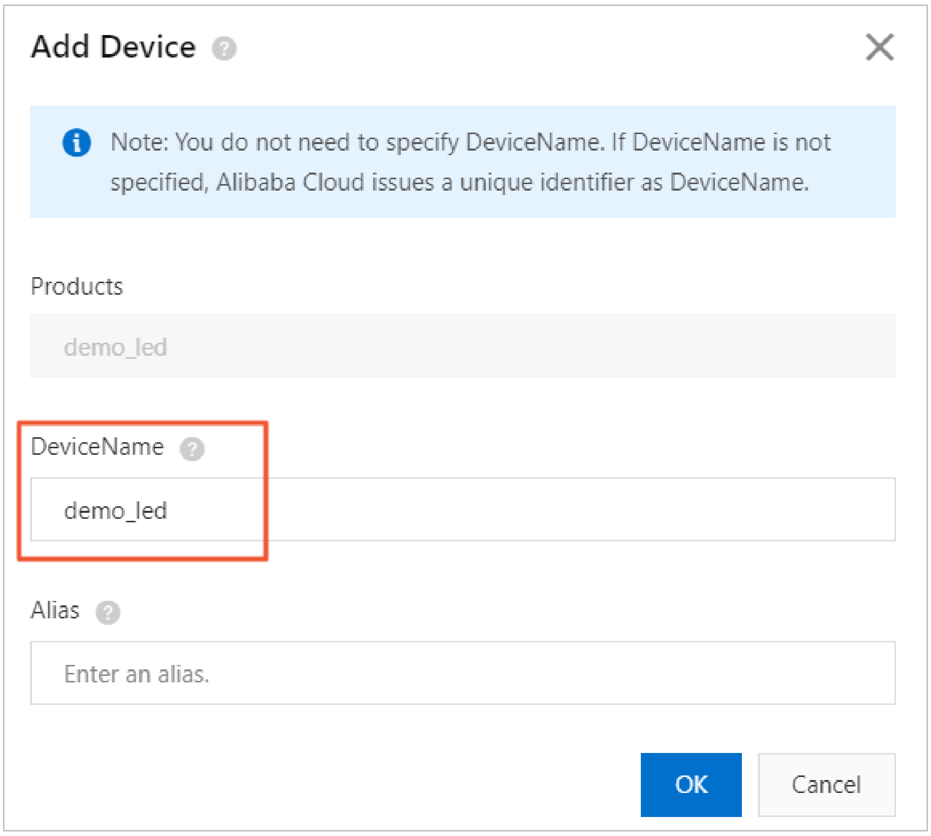
To obtain the value of Node Name, find the high_temperature event of the simulated LED light in UaExpert and click the event name. Check the value of DisplayName in the **Attributes** section on the right and use it as the value of Node Name.



Attribute	Value
NodeId	NodeId
NamespaceIndex	2
IdentifierType	Numeric
Identifier	6
NodeClass	ObjectType
BrowseName	2, "high_temperature"
DisplayName	"", "high_temperature"
Description	"", "high_temperature"
WriteMask	0
UserWriteMask	0
IsAbstract	false

Reference	Target DisplayName
HasProperty	temperature

3. Add the simulated LED light as a sub-device to the OPC UA product. For more information, see Create a device.



Add Device

Note: You do not need to specify DeviceName. If DeviceName is not specified, Alibaba Cloud issues a unique identifier as DeviceName.

Products

demo_led

DeviceName

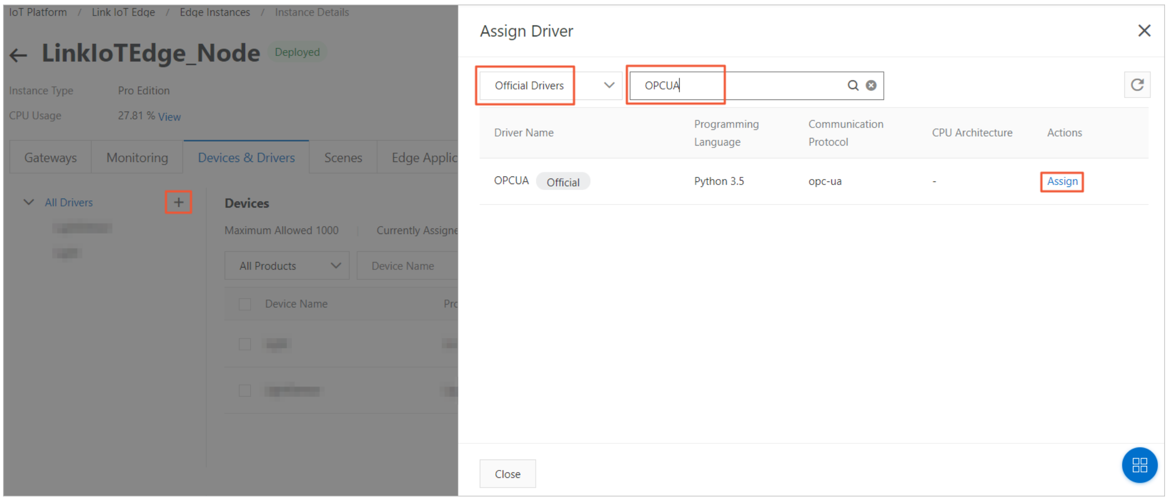
demo_led

Alias

Enter an alias.

Step 4: Configure the edge instance

1. On the homepage of the IoT Platform console, choose **Link IoT Edge** > **Edge Instances** in the left-side navigation pane. On the **Edge Instances** page, find the target edge instance and click **View** in the **Actions** column.
2. On the **Instance Details** page that appears, click the **Devices & Drivers** tab and then click **Assign Driver**. In the **Assign Driver** dialog box that appears, select **Official Drivers**, find the driver named OPCUA, and then click **Assign** in the **Actions** column.



IoT Platform / Link IoT Edge / Edge Instances / Instance Details

LinkIoTEdge_Node Deployed

Instance Type Pro Edition

CPU Usage 27.81 % View

Devices

Maximum Allowed 1000 | Currently Assigne

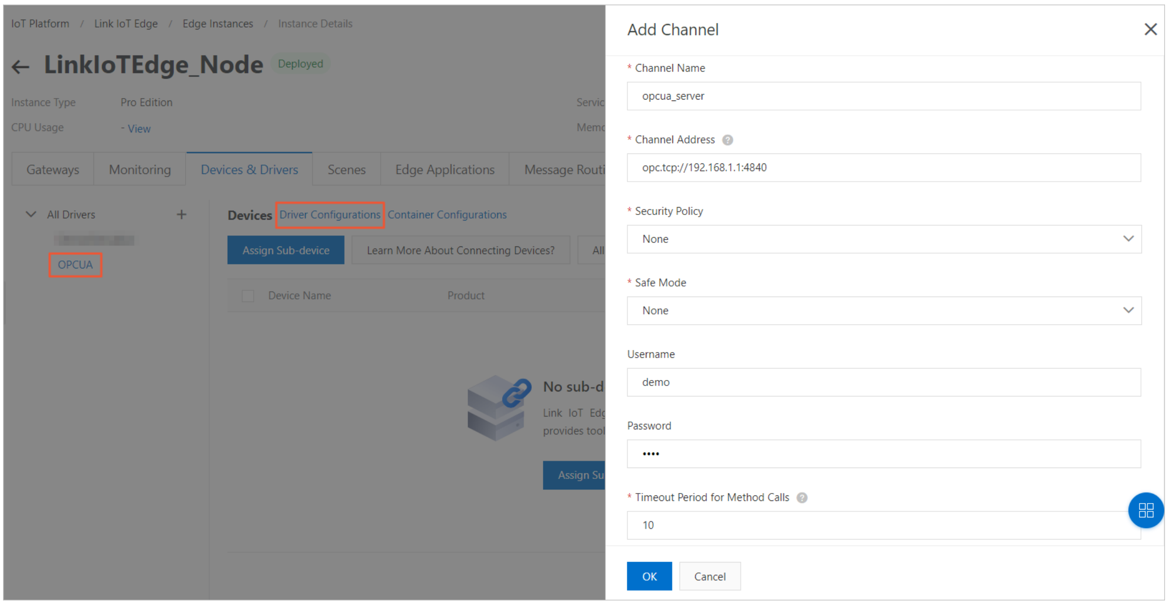
Assign Driver

Official Drivers

OPCUA

Driver Name	Programming Language	Communication Protocol	CPU Architecture	Actions
OPCUA Official	Python 3.5	opc-ua	-	Assign

3. Go back to the **Devices & Drivers** tab, click **OPCUA** in the **All Drivers** section on the left, and then click **Driver Configurations** next to **Devices** on the right. In the **Driver Configurations** dialog box that appears, click **Add Channel**. In the **Add Channel** dialog box that appears, set channel parameters and click **OK**.



Add Channel

* Channel Name: opcua_server

* Channel Address: opc.tcp://192.168.1.1:4840

* Security Policy: None

* Safe Mode: None

Username: demo

Password: ••••

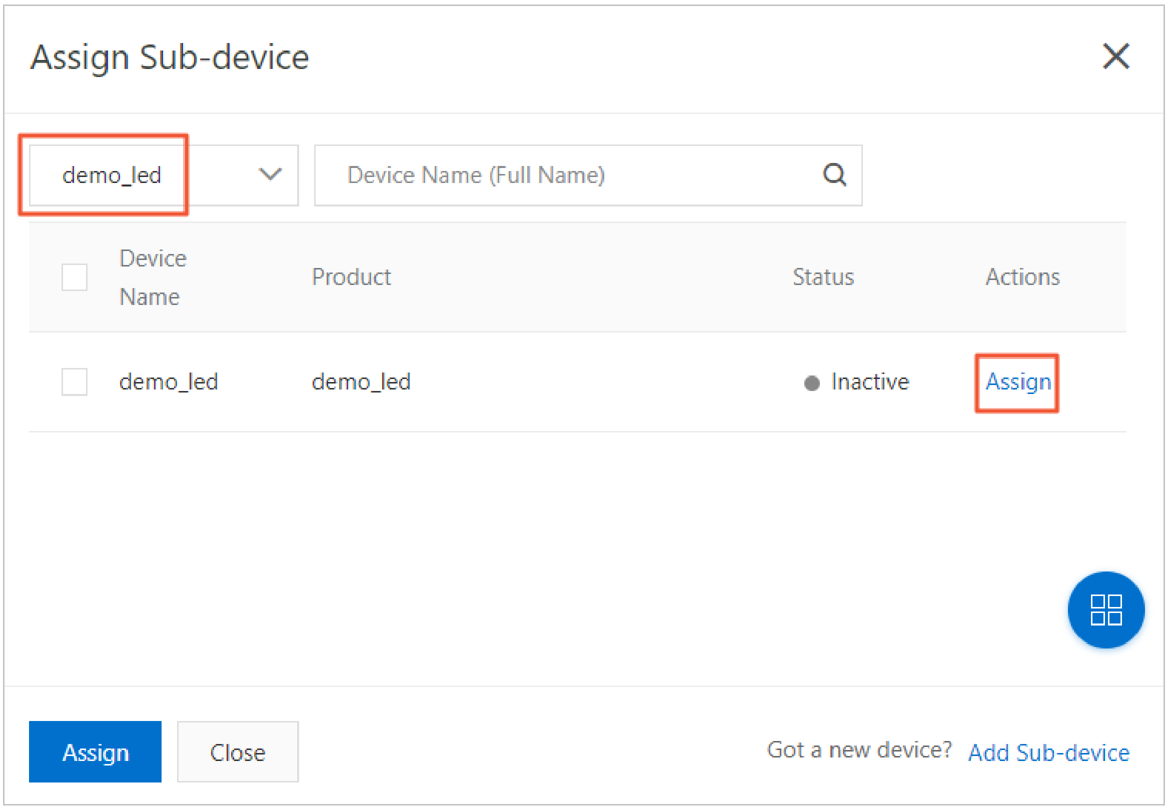
* Timeout Period for Method Calls: 10

OK Cancel

Description of channel parameters

Parameter	Description	Example
Channel Name	The name of the OPC UA channel.	*opcua_server*
Channel Address	The URL of the OPC UA Server.	*opc.tcp://192.168.1.1:4840*
Username	The username for connecting to the OPC UA Server.	*demo*
Password	The password for connecting to the OPC UA Server.	*abc123*
Timeout Period for Method Calls	The timeout period of a request for calling the OPC UA Server.	*10*

4. On the **Devices & Drivers** tab, click **OPCUA** in the **All Drivers** section on the left and click **Assign Sub-device** under **Devices**. In the **Assign Sub-device** dialog box that appears, find the simulated LED light and click **Assign** in the **Actions** column.

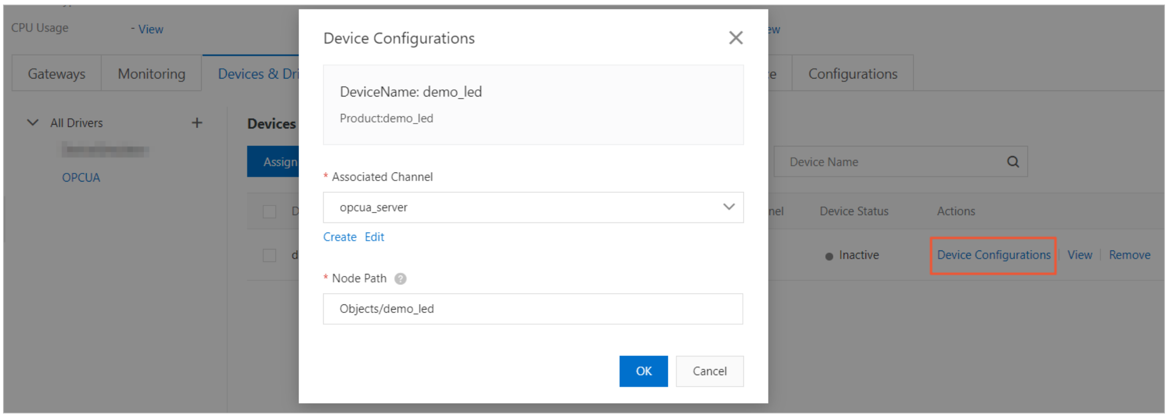


Assign Sub-device

demo_led | Device Name (Full Name)

□	Device Name	Product	Status	Actions
□	demo_led	demo_led	● Inactive	Assign

5. Go back to the **Devices & Drivers** tab, find the simulated LED light you assigned, and then click **Device Configurations** in the **Actions** column. The **Device Configurations** dialog box appears.



Device Configurations

DeviceName: demo_led
Product:demo_led

* Associated Channel: opcua_server

* Node Path: Objects/demo_led

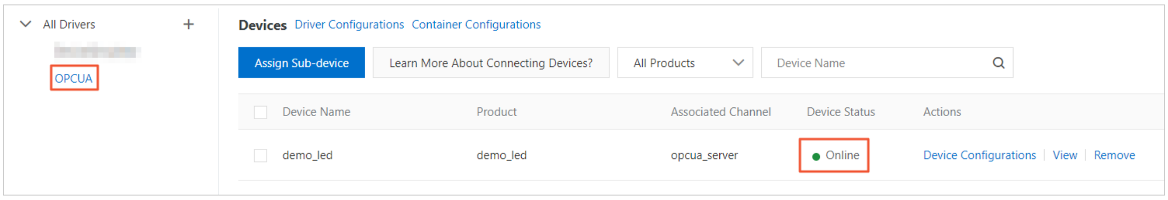
Description of parameters in the Device Configurations dialog box

Parameter	Description
Associate Channel	Select the channel you added.
Node Path	Set this parameter to the absolute path of the simulated LED light on the OPC UA Server. The path starts with Objects. In this example, the absolute path is *Objects/demo_led*.

6. On the **Instance Details** page of the edge instance, click **Deploy** in the upper-right corner to deploy the edge instance.

7. On the **Instance Details** page, click the **Devices & Drivers** tab. Click **OPCUA** in the **All Drivers**

section on the left, find the simulated LED light on the right, and then check whether the value of Device Status is Online.



□	Device Name	Product	Associated Channel	Device Status	Actions
□	demo_led	demo_led	opcua_server	● Online	Device Configurations | View | Remove