



**PUBLIC**

Document Version: 4.84.0 – 2024-03-26

# **Internet of Things Gateway**

## **SAP IoT services for SAP BTP**

# Content

- 1 Introduction. . . . . 5**
- 1.1 Internet of Things Gateway Cloud and Internet of Things Edge Platform. . . . . 5
- 1.2 Measure and Command. . . . . 6
  
- 2 Onboard Devices. . . . . 7**
- 2.1 Manual Onboarding. . . . . 7
- 2.2 Automatic Onboarding. . . . . 7
- 2.3 Access the Device Onboarding Certificate. . . . . 7
- 2.4 Register a Device. . . . . 8
- 2.5 Request a Device Certificate. . . . . 9
- 2.6 Register a Sensor. . . . . 10
- 2.7 Device Vitality Algorithm. . . . . 11
  
- 3 Internet of Things Gateway Cloud. . . . . 13**
- 3.1 MQTT. . . . . 13
  - Measure Message Format (JSON). . . . . 13
  - Report the Measure Processing Results (JSON). . . . . 23
  - Command Message Format (JSON). . . . . 25
  - Device Authorization and Authentication. . . . . 26
  - Endpoint Structure. . . . . 28
  - Command Management. . . . . 28
- 3.2 REST. . . . . 29
  - Measure Message Format (JSON). . . . . 29
  - Report the Measure Processing Results (JSON). . . . . 39
  - Measure Message Format (Protobuf). . . . . 42
  - Report the Measure Processing Results (Protobuf). . . . . 44
  - Command Message Format (JSON). . . . . 47
  - Command Message Format (Protobuf). . . . . 48
  - Device Authorization and Authentication. . . . . 49
  - Endpoint Structure. . . . . 50
  
- 4 Internet of Things Edge Platform. . . . . 51**
- 4.1 Network Requirements. . . . . 51
- 4.2 OS Requirements. . . . . 52
- 4.3 Audit Log. . . . . 53
- 4.4 Offline Operations. . . . . 54
  - Configuration. . . . . 55
  - Access using Swagger. . . . . 57

	Buffering on Internet of Things Edge Platform. . . . .	57
4.5	MQTT. . . . .	59
	Measure Message Format (JSON). . . . .	59
	Report the Measure Processing Results (JSON). . . . .	70
	Measure Payload Customization. . . . .	73
	Command Message Format (JSON). . . . .	74
	Device Authorization and Authentication. . . . .	75
	Endpoint Structure. . . . .	76
4.6	REST. . . . .	81
	Measure Message Format (JSON). . . . .	81
	Report the Measure Processing Results (JSON). . . . .	92
	Measure Payload Customization. . . . .	95
	Command Message Format (JSON). . . . .	97
	Measure Message Format (Protobuf). . . . .	98
	Report the Measure Processing Results (Protobuf). . . . .	101
	Command Message Format (Protobuf). . . . .	103
	Device Authorization and Authentication. . . . .	104
	Endpoint Structure. . . . .	105
4.7	CoAP. . . . .	111
	Overview. . . . .	111
	Adapter Configuration. . . . .	111
4.8	FILE. . . . .	114
	Overview. . . . .	114
	Adapter Configuration. . . . .	114
4.9	Modbus. . . . .	115
	Overview. . . . .	116
	Adapter Configuration. . . . .	117
	Modbus Function Codes. . . . .	120
	Node and Data Item Addressing. . . . .	120
4.10	OPC UA. . . . .	121
	Overview. . . . .	121
	Adapter Configuration. . . . .	123
	Device Model Mapping. . . . .	126
	Mapping Business Objects in the Internet of Things Edge Platform OPC UA. . . . .	128
4.11	SigFox. . . . .	133
	Overview. . . . .	133
	Architecture. . . . .	134
	Adapter Configuration. . . . .	135
	Security. . . . .	140
	Device Features. . . . .	141
	APIs. . . . .	142

	Cloud Entities. . . . .	142
	Device Registration. . . . .	142
4.12	SNMP. . . . .	143
	Overview. . . . .	144
	Adapter Configuration. . . . .	144
	Network Addressing. . . . .	145
	Node Discovery. . . . .	146
<b>5</b>	<b>Internet of Things Edge Platform Scalability. . . . .</b>	<b>147</b>
5.1	Architectural Overview. . . . .	147
	Limitations. . . . .	147
	Generic Aspects. . . . .	148
	MQTT Overview. . . . .	149
	REST Overview. . . . .	150
5.2	Internet of Things Edge Platform Configuration. . . . .	150
5.3	MQTT-Specific Configuration. . . . .	154
5.4	REST-Specific Configuration. . . . .	157
5.5	Deployment of Custom Bundles. . . . .	157
5.6	Internet of Things Edge Platform Upgrade. . . . .	158
5.7	Internet of Things Edge Platform Startup. . . . .	159
5.8	Networking Setup. . . . .	160

# 1 Introduction

## 1.1 Internet of Things Gateway Cloud and Internet of Things Edge Platform

The Internet of Things Gateway Cloud and the Internet of Things Edge Platform are software components that directly interact with devices. It both collect data from and send commands to them and provides adapters for the communication with various protocols.

- The Internet of Things Gateway deployed in the cloud is a managed component. Each Internet of Things Service comes with an Internet of Things Gateway for MQTT and REST deployed in the cloud.
- The Internet of Things Edge Platform is a component that is managed by the customer.

The Internet of Things Gateway Cloud and the Internet of Things Edge Platform can be deployed with various adapters that support the following list of protocols:

Protocol	Cloud	Edge
MQTT	x	x
HTTP (REST)	x	x
CoAP		x
File		x
Modbus		x
OPC UA		x
SigFox		x
SNMP		x

## 1.2 Measure and Command

### Measure

A measure describes sensor data that is sent from the device to the Internet of Things Gateway Cloud or Internet of Things Edge Platform.

It is a representation of a capability that contains one or multiple properties. A measure must always be wrapped in a measure message that contains additional metadata, such as the `capabilityAlternateId` and `sensorAlternateId`. The metadata is used for the validation of the measure.

### Command

A command describes data that is sent from the Internet of Things Gateway Cloud or Internet of Things Edge Platform to the device.

It is a representation of a capability that contains one or multiple properties. A command must always be wrapped in a command message that contains additional metadata, such as the `capabilityAlternateId` and the `sensorAlternateId`.

For more information about the measures and commands related to the device model, please refer to section in the *Introduction*.

## 2 Onboard Devices

You can onboard devices to the Internet of Things Service either automatically or manually.

### 2.1 Manual Onboarding

If you onboard devices manually, this allows you to provide an device instance on the platform even before the device itself connects.

Manual onboarding relies on dedicated APIs that you can use to create the device in the Internet of Things Service. When the device connects to the platform for the first time, the data it generates is associated to the device onboarded manually, and no other instance is created on the platform. After a device has been onboarded, you can download the device certificate to connect securely to the Internet of Things Gateway Cloud MQTT and REST.

For more information, please refer to the tutorial .

### 2.2 Automatic Onboarding

The process of obtaining a device certificate is a two-step process. In the first step, the physical device must obtain a registration certificate. Using this registration certificate, you can then create a device in the Internet of Things Service. Once this device entity exists, the physical device can request its device-specific certificate. The following steps provide an overview of the process:

- Physical device must access its onboarding certificate.
- Physical device registers itself by creating a device entity in the Internet of Things Service data model.
- Physical device requests a device certificate for the created device entity.
- Physical device is registered completely and can authenticate itself using the device certificate when sending measures.

The detailed process is described in the following sections. For additional information about each API, please refer to the [Internet of Things Service API](#) documentation.

### 2.3 Access the Device Onboarding Certificate

A device needs an onboarding certificate to register itself.

Onboarding certificates are valid in the context of a single gateway. A device can register itself for the correct gateway corresponding to the onboarding certificate. Consequently, the onboarding certificate can only be

retrieved in the context of a single specific gateway within a tenant. The onboarding certificate can be used by multiple devices to execute the onboarding procedure. This allows the creator of devices to retrieve the onboarding certificate during the time of device creation and add it to the software running on the physical device. Once the physical device is delivered to a customer, the device can continue the onboarding procedure automatically.

The Internet of Things Service offers a set of APIs to retrieve the onboarding certificate. This API is protected by standard user credentials using Basic Auth. For more information, please refer to section in the *Security* documentation.

The following table lists the endpoints for accessing the onboarding certificate as a P12 or PEM file.

HTTP Attribute	Description	Path using base path: /iot/core/api/v1
GET	Downloads a device registration P12 file	/tenant/{tenantId}/gateways/{gatewayId}/deviceRegistrations/clientCertificate/p12
GET	Downloads a device registration PEM file	/tenant/{tenantId}/gateways/{gatewayId}/deviceRegistrations/clientCertificate/pem

### Note

When downloading a registration certificate, you receive a key pair including an encrypted private key in addition to the certificate itself. If you lose this key pair, we are not able to recover the private key. Instead, a new key pair has to be generated.

## 2.4 Register a Device

When a device has received an onboarding certificate, it can access the Internet of Things Service API to create a new device entity.

The following table specifies the respective API. This API is protected through a device registration certificate in addition to Basic Auth user credentials.

HTTP Attribute	Description	Path using base path: /iot/core/api/v1
POST	Creates a device	/tenant/{tenantId}/devices



## 2.5 Request a Device Certificate

Devices can request device certificates in different ways.

They can obtain the certificate in the PEM or the P12 format depending on their specific requirements. Furthermore, there are two different approaches for generating PEM certificates: using a certificate signing request (CSR) generated and provided by the device or without such a CSR. The following table lists the three API endpoints to obtain a device certificate.

HTTP Attribute	Description	Path using base path: /iot/core/api/v1
GET	Downloads device p12 file	/tenant/{tenantId}/devices/{deviceId}/authentications/clientCertificate/p12
GET	Downloads a device private key and certificate in PEM format	/tenant/{tenantId}/devices/{deviceId}/authentications/clientCertificate/pem
POST	Creates a device certificate in PEM format	/tenant/{tenantId}/devices/{deviceId}/authentications/clientCertificate/pem

The `POST` request for creating a PEM certificate expects a payload containing a certificate signing request and a specification of the type of certificate, that is `clientCertificate`.

### Sample Code

```
{
  "csr": "string",
  "type": "clientCertificate"
}
```

A CSR must specify a common name (CN) according to the following structure:

### Sample Code

```
CN=deviceAlternateId:<...>|gatewayId:<...>|tenantId:<...>|instanceId:<...>
```

For more information, please refer to section [Certificate Structure \[page 27\]](#)

### Note

For security reasons, we strongly recommend that you generate the private key on the physical device itself by using an RSA key pair size of at least 2,048 bits. Furthermore, the CSR must be signed using the private key and SHA256withRSA algorithm.

## Sample Certificate Signing Request

### ↔ Sample Code

```
[PKCS #10 certificate request:
Sun RSA public key, 2048 bits
  modulus:
205828706690693411223759677788530418050800140582549447673127360940926675544
409309358103836291808914517733332902365867938289058885312647576040883678315
072006497525988260600687443687096083377882583430133792340275149279349798549
253061579693130164145846748543035413540891465601308995036513659210110456496
042586944742501492057645806295747850374077863740826856709666562544725198043
644159170459500135407584264024668345698221005298153604449903771862519146627
289346166011092949270947741419636931424093004050568132183277124558841357661
580550366382618513911183078457942273923189811605492308685610371412893244710
96297681743102299
  public exponent: 65537 subject: <CN=deviceAlternateId:1234562599|
gatewayId:3|tenantId:1756591794|instanceId:iot-certificate, OU=IoT
Services, O=SAP Trust Community, L=, ST=, C=DE>
  attributes: 0
]
```

When the device has received the respective device certificate, it can start sending measure data using this certificate for authentication.

In addition, for security reasons, delete the onboarding certificate after the device certificate has been created successfully.

## 2.6 Register a Sensor

After a device has received a device certificate, it can access the Internet of Things Service API to create a new sensor entity.

The following table specifies the respective API. This API is protected by means of a device certificate in addition to Basic Auth user credentials.

HTTP Attribute	Description	Path using base path: /iot/ core/api/v1
POST	Creates a sensor	/tenant/{tenantId}/sensors

## 2.7 Device Vitality Algorithm

### Algorithm 1: MQTT protocol with device certificate authorization

A device is considered online when an MQTT connection is active for this device.

#### Note

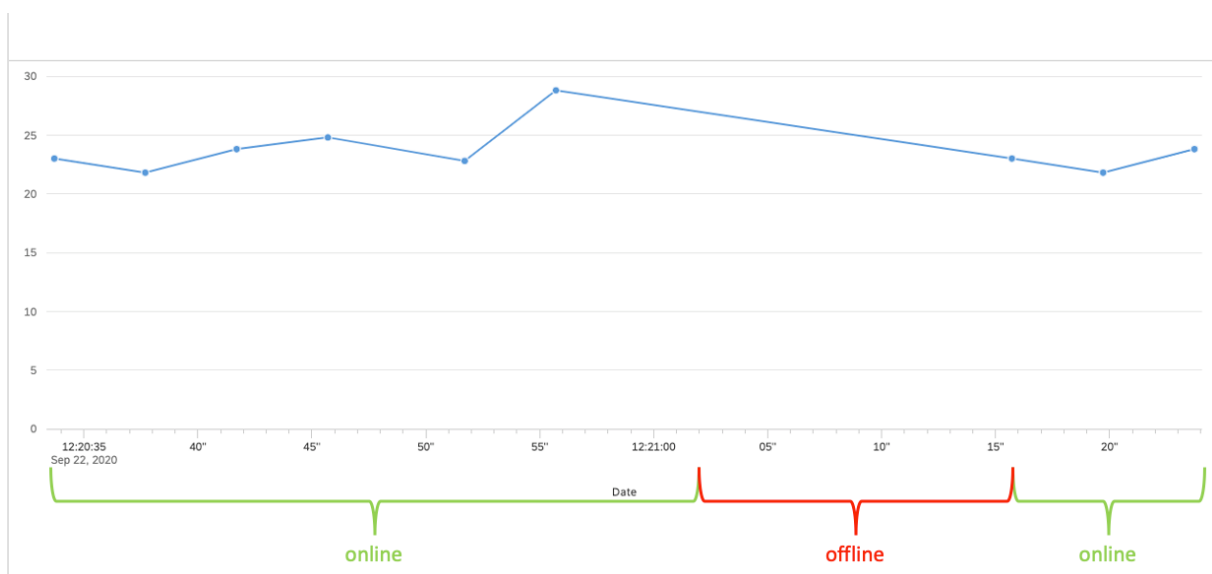
Update interval is 3 seconds and the eventual changes in this time range will be lost.

### Algorithm 2: OPC UA protocol and fallback implementation for MQTT protocol (when devices are connected using a router device certificate or without certificates on the edge)

The Internet of Things Gateway will calculate the expected frequency based on the last 60 measures received by a device. If the device keeps the same rate for measure ingestion (a 50% tolerance is applied to the estimated rate), then it is considered online, otherwise it will be considered offline.

Online state will be automatically updated by the respective gateway.

The following image shows the device status overlying on the data visualization chart from the Internet of Things Service Cockpit.



### Algorithm 3: REST protocol

If a device has not sent a measure for more than 3 seconds, it is considered offline. The online state will be automatically updated by the respective gateway.

### Matrix table for algorithm usage

	MQTT	REST	OPC UA
Device connected using Device Certificate (Cloud/Edge)	Algorithm 1	Algorithm 3	
Device connected as Router Device via Internet of Things Gateway Cloud	Algorithm 2	Algorithm 3	
Device connected via Internet of Things Edge Platform	Algorithm 2	Algorithm 3	Algorithm 2

## 3 Internet of Things Gateway Cloud

Each Internet of Things Service comes with an Internet of Things Gateway for MQTT and REST deployed in the cloud.

### Related Information

[MQTT \[page 13\]](#)

[REST \[page 29\]](#)

### 3.1 MQTT

The Internet of Things Gateway Cloud for MQTT provides interfaces that are used by devices to send measures and receive commands. To send measurement data, the device must publish data in the JSON format to topic `measures/<device alternate ID>`. It may also receive notifications about the processing status of the published measures. In order to receive such notifications, the device should subscribe to the `ack/<device alternate ID>` topic. Finally, to receive commands from the cloud, the device should subscribe to the topic `commands/<device alternate ID>`. The following sections describe the corresponding message formats.

### Related Information

[Measure Message Format \(JSON\) \[page 13\]](#)

[Report the Measure Processing Results \(JSON\) \[page 23\]](#)

[Command Message Format \(JSON\) \[page 25\]](#)

[Device Authorization and Authentication \[page 26\]](#)

[Endpoint Structure \[page 28\]](#)

[Command Management \[page 28\]](#)

#### 3.1.1 Measure Message Format (JSON)

After a device has been successfully registered and a certain sensor has been assigned, the device can start sending messages that conform to a capability defined for the respective sensor type.

The schema of a valid measure message format looks as follows:

## Sample Code

```
{
  "oneOf": [
    {
      "$ref": "#/definitions/messageObject"
    },
    {
      "type": "array",
      "items": {
        "$ref": "#/definitions/messageObject"
      }
    }
  ],
  "definitions": {
    "messageObject": {
      "title": "IoT Gateway JSON Message Schema",
      "type": "object",
      "properties": {
        "sensorAlternateId": {
          "id": "sensorAlternateId",
          "type": "string",
          "required": true,
          "description": "alternate id of the sensor the measures belong
to"
        },
        "capabilityAlternateId": {
          "id": "capabilityAlternateId",
          "type": "string",
          "required": true,
          "description": "alternate id of the capability the measures
conform to"
        },
        "timestamp": {
          "oneOf": [
            {
              "id": "timestamp",
              "type": "string",
              "required": false,
              "description": "time of the measure retrieval at the
device in ISO 8601 format"
            },
            {
              "id": "timestamp",
              "type": "number",
              "required": false,
              "description": "time of the measure retrieval at the
device in Unix format in milliseconds since January, 1st 1970 "
            }
          ]
        },
        "measureMessageId": {
          "id": "measureMessageId",
          "type": "string",
          "required": false,
          "description": "an identifier of the message with measurement
data, unique within the device"
        },
        "processingTag": {
          "id": "processingTag",
          "type": "string",
          "required": false,
          "description": "property for dispatching measures to the
corresponding Message Selector"
        },
        "measures": {
          "id": "measures",
```

```

        "type": "array",
        "minItems": 1,
        "required": true,
        "items": {
          "oneOf": [
            {
              "type": "array",
              "minItems": 1,
              "required": true,
              "items": {
                "type": "string",
                "description": "list of measures that conform to
the capability definition with the given alternate id"
              }
            },
            {
              "type": "object",
              "minItems": 1,
              "required": true,
              "description": "list of measures that conform to the
capability definition with the given alternate id"
            }
          ]
        }
      }
    }
  }
}

```

### 3.1.1.1 Single Measures

The schema of a valid message with a single measure that is sent to the Internet of Things Gateway cloud looks as follows:

#### Sample Code

```

{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "measureMessageId": "123",
  "measures": [
    {
      "Humidity": 90,
      "Temperature": 23.5,
      "Status": "Good"
    }
  ]
}

```

The payload includes a message that conforms to a capability with the alternate ID `capability1`. The measures are for the sensor with the alternate ID `sensor1`. The measure contains three properties, namely `Humidity`, `Temperature`, and `Status`. These fields are part of the respective capability definition.

It is possible to send the measure properties in any order and leave out values that are unknown.

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "measures": [
    {
      "Status": "Good",
      "Humidity": 90
    }
  ]
}
```

## Timestamp

For each measure, the timestamp can be reported. The Internet of Things Service use timestamp to order the measurement samples returned by the Internet of Things Service API. If a timestamp is not specified in the ingestion payload, the time of arrival of the message will be referenced.

The timestamp can be provided in two ways:

- **Message Timestamp**

As described in the schema above, you can specify the ingestion time for the whole message through the property `timestamp`.

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "timestamp": 1413191650000,
  "measures": [
    {
      "Humidity": 90,
      "Temperature": 23.5
    }
  ]
}
```

Both formats, ISO 8601 and a Unix time in milliseconds from epoch (that is a 13-digit number), are allowed.

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "timestamp": "2017-05-30T15:06:15.123Z",
  "measures": [
    {
      "Humidity": 90,
      "Temperature": 23.5
    }
  ]
}
```



- **Measure Timestamp**

- If you have modeled a capability with a property of type `date`, you can use it to specify the measure timestamp (that is the timestamp of the sample).

#### Sample Code

```
{
  "sensorAlternateId": "sensor2",
  "capabilityAlternateId": "capability2",
  "measures": [
    {
      "Pressure": 90,
      "Temperature": 19.3,
      "Timestamp": 1413191650000
    }
  ]
}
```

Both formats, ISO 8601 and a Unix time in milliseconds from epoch (that is a 13-digit number), are allowed.

#### Sample Code

```
{
  "sensorAlternateId": "sensor2",
  "capabilityAlternateId": "capability2",
  "measures": [
    {
      "Pressure": 90,
      "Temperature": 19.3,
      "Timestamp": "2017-05-30T15:06:15.123Z"
    }
  ]
}
```

Looking at the example above, the payload includes a message that conforms to a capability with alternate ID `capability2`. The measures are for the sensor with the alternate ID `sensor2`. The measure contains three properties, namely `Pressure`, `Temperature`, and `Timestamp` (with the property `Timestamp` of type `date`). These fields are part of the respective capability definition. Modeling a property of type `date` is useful if:

- You want to use one of the properties of the device as timestamp of the measure.
- You want to send a single message with batched measures, for example, a message containing more samples for the capability, each one with a different timestamp (as for a default time series). For more information, please refer to section [Batched Measures \[page 19\]](#).
- You can use a special property named `'_time'` and not modelled into the capability to specify the measure timestamp (that is the timestamp of the sample).

#### Sample Code

```
{
  "sensorAlternateId": "sensor3",
  "capabilityAlternateId": "capability3",
  "measures": [
    {
      "Pressure": 90,
      "Temperature": 19.3,
      "_time": 1413191650000
    }
  ]
}
```

```
} ]
```

Both formats, ISO 8601 and a Unix time in milliseconds from epoch (that is a 13-digit number), are allowed.

### Sample Code

```
{
  "sensorAlternateId": "sensor3",
  "capabilityAlternateId": "capability3",
  "measures": [
    {
      "Pressure": 90,
      "Temperature": 19.3,
      "_time": "2017-05-30T15:06:15.123Z"
    }
  ]
}
```

Looking at the example above, the payload includes a message that conforms to a capability with alternate ID `capability3`. The measures are for the sensor with the alternate ID `sensor3`. The measure contains three properties, namely `Pressure`, `Temperature`, and `_time`, but only `Pressure` and `Temperature` are part of the respective capability definition.

Using the special `_time` property is useful if:

- You want to use it as timestamp of the measure, without modelling it as a property of the capability.
- You want to send a single message with batched measures, for example, a message containing more samples for the capability, each one with a different timestamp (as for a default time series). For more information, please refer to section [Batched Measures \[page 19\]](#).

### Note

- Both, the Message Timestamp and the Measure Timestamp are normalized to Unix time in milliseconds from epoch (that is a 13-digit number) after the ingestion and represented as long.
- The Measure Timestamp has a higher priority than the Message Timestamp, for example, if both are present, the platform uses the Measure Timestamp as ingestion timestamp.
- The Measure Timestamp expressed via the special property `"_time"` has a higher priority to the Measure Timestamp expressed via a property of type `'date'`; for example, if both are present, the platform uses the value of the `_time` property as ingestion timestamp.

## Processing Tag

A message can be tagged to be dispatched to the desired message selector. To do this, after have configured the processing of messages through the Message Processing API, you can specify the `processingTag` property in the message. Further information about the Internet of Things Message Processing, selectors, and their configuration can be found in section in the *Internet of Things Message Processing* documentation.

The following sample payload relies on a message selector using the `processingTag` property with value `sampleTag`.

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "processingTag": "sampleTag",
  "measures": [
    {
      "Status": "Good",
      "Humidity": 90
    }
  ]
}
```

## 3.1.1.2 Batched Measures

It is possible to send multiple measures that conform to the same capability in a single message.

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "measures": [
    {
      "Humidity": 90,
      "Temperature": 23.5,
      "Status": "Good"
    },
    {
      "Humidity": 93,
      "Temperature": 24.0,
      "Status": "Bad"
    }
  ]
}
```

## Timestamp

- Message Timestamp

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "timestamp": 1413191650000,
  "measures": [
    {
      "Humidity": 90,
      "Temperature": 23.5,
      "Status": "Good"
    }
  ],
}
```

```

    {
      "Humidity":93,
      "Temperature":24.0,
      "Status":"Bad"
    }
  ]
}

```

If you specify the timestamp as message timestamp, it will be applied to all the samples (for example, in this case both, the first and the second samples, will have the same ingestion timestamp 1413191650000)

- **Measure Timestamp**

- If you have modeled a property as type `date`, its value will result in the ingestion timestamp of each sample on the platform.

#### ↔ Sample Code

```

{
  "sensorAlternateId":"sensor2",
  "capabilityAlternateId":"capability2",
  "measures":[
    {
      "Pressure":90,
      "Temperature":19.3,
      "Timestamp": 1413191650000
    },
    {
      "Pressure":75,
      "Temperature":10.1,
      "Timestamp": 1413191790000
    }
  ]
}

```

For example, in this case you have two samples, the first one with 1413191650000 as ingestion timestamp and the second one with 1413191790000.

- If you have specified a `_time` property, without having modeled it as a property of the capability, its value will result in the ingestion timestamp of each sample on the platform.

#### ↔ Sample Code

```

{
  "sensorAlternateId":"sensor3",
  "capabilityAlternateId":"capability3",
  "measures":[
    {
      "Pressure":90,
      "Temperature":19.3,
      "_time": 1413191650000
    },
    {
      "Pressure":75,
      "Temperature":10.1,
      "_time": 1413191790000
    }
  ]
}

```

For example, in this case you have two samples, the first one with 1413191650000 as ingestion timestamp and the second one with 1413191790000.

### 3.1.1.3 Compressed Single Measures

It is possible to compress the measures by omitting the field names. In this case, the measures array is an array of arrays and the order must match the order of the properties in the capability definition.

#### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "measures": [
    [
      90,
      23.5,
      "Good"
    ]
  ]
}
```

It is also possible to send null values for properties for which no values exist.

#### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "measures": [
    [
      90,
      null,
      "Good"
    ]
  ]
}
```

### 3.1.1.4 Compressed Batched Measures

It is possible to send multiple measures with the compressed format (array) in a single message.

#### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "measures": [
    [
      90,
      23.5,
      "Good"
    ],
    [
      94,
      24.0,
      "Bad"
    ]
  ]
}
```

```
]
```

### 3.1.1.5 Batched Measure Messages

It is possible to send a batch of messages that conform to different capabilities. In this case, the batch contains an array of messages.

#### Sample Code

```
[
  {
    "sensorAlternateId": "sensor1",
    "capabilityAlternateId": "capability1",
    "measures": [
      {
        "Humidity": 90,
        "Temperature": 23.5,
        "Status": "Good"
      },
      {
        "Humidity": 93,
        "Temperature": 24.0,
        "Status": "Bad"
      }
    ]
  },
  {
    "sensorAlternateId": "sensor2",
    "capabilityAlternateId": "capability2",
    "measures": [
      {
        "Pressure": 90,
        "Temperature": 19.3,
        "Timestamp": 1413191650000
      }
    ]
  }
]
```

### 3.1.1.6 Measures for the Auto-Onboarding of Sensors

It is possible to auto-onboard sensors during the data ingestion. Auto-onboarding means that sensors that do not exist are created automatically during data ingestion. In this case, an additional attribute, namely the `sensorTypeAlternateId`, must be passed. If the sensor with the given `sensorAlternateId` does not exist, a new sensor is created for the device that is assigned to the sensor type with this `sensorTypeAlternateId`.

#### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "sensorTypeAlternateId": "12",
  "capabilityAlternateId": "capability1",
```

```
"measures": [
  {
    "Humidity": 90,
    "Temperature": 23.5,
    "Status": "Good"
  }
]
```

#### Note

Only numeric values are supported for the alternate ID of a sensor type.

## 3.1.2 Report the Measure Processing Results (JSON)

The Internet of Things Gateway reports the processing results for the posted measures to the device by publishing corresponding records to the topic `ack/<device alternate ID>`.

Note that the Internet of Things Gateway reports the processing status immediately, before forwarding the measures to the Internet of Things Service. If the Internet of Things Gateway considers the measure message invalid and rejects it, this outcome is final. However, if the Internet of Things Gateway accepts the measure message as valid, it may still be rejected or lost further down the road. Thus, the Internet of Things Gateway typically does not return the `200 OK` status code, but only `202 Accepted`. The MQTT API will be extended in future releases of the gateway, allowing HTTP clients to query the final processing outcome of measure messages posted earlier, including the processing by the Internet of Things Service.

### 3.1.2.1 Response Message

The detailed status message is a list of objects. The number of objects is either one or corresponds to the number of measure messages posted within [Batched Measure Messages \[page 22\]](#). In the latter case, the order of the objects is the same as the order of the measure messages in the batch. Each object can have the following fields:

```
{
  "id": "optional; the value of the measureMessageId field if the device has
  provided it when posting the measurement data",
  "code": "the status code for this Measure Message; the same as the HTTP status
  code if there is only one Measure Message in the HTTP request",
  "sensorAlternateId": "value from the request",
  "capabilityAlternateId": "value from the request",
  "messages": [a list of human-readable strings, mostly error messages, generated
  during the verification of the measure message]
}
```

Subsequent versions of the Internet of Things Gateway may extend the object with additional fields. All fields except for the `code` field can be missing (for example, if the JSON request could not be parsed, or no errors were detected).

If the device needs to bind the objects in the response message to the corresponding measure messages explicitly, it should provide unique values of `measureMessageId` field for every measure message. These values are returned in the `id` field of the matching response messages.

### 3.1.2.2 Processing Status Codes

The semantics of the `code` field is the same as of the corresponding HTTP status code (see the following table).

Status code	Meaning
200 OK	The Internet of Things Gateway has received a confirmation from the Internet of Things Service that the measure message was processed correctly before returning the result to the device. Almost never occurs.
202 Accepted	The measure message has passed the validation check executed by the Internet of Things Gateway.
400 Bad Request	The measure message has failed the validation check executed by the Internet of Things Gateway. If the message posted is a batched message, all message parts have failed the validation check. All measures were discarded. The body of the response contains detailed information.
401 Unauthorized	Authentication has failed, or the authenticated device is not authorized to post measures on behalf of the device with the alternate ID specified in the URL. For more information, please refer to section <a href="#">Device Authorization and Authentication [page 26]</a> .
500 Internal Server Error	The Internet of Things Gateway is not configured correctly, or a software error has occurred in the Internet of Things Gateway.

### 3.1.2.3 Error Messages

The `messages` field can contain any human-readable error messages. There is no fixed list of allowed values, but the messages may start with the strings listed in the following table.



Message	Condition
Unauthorized	The client is not authorized to post the measure messages on behalf of the device. In particular, this may mean an HTTP authentication failure.
Invalid device alternate ID	Most likely to appear when a valid and authenticated router device posts measure messages on behalf of another one.
Invalid message format	This means that the whole payload could not be parsed, for example due to invalid JSON syntax. The sensor and device alternate IDs, etc., cannot be extracted in this case.
Invalid sensor alternate ID	The sensor does not exist and could not be onboarded automatically.
Invalid capability alternate ID	The capability does not exist for the sensor, or it does not accept measures (being of the command type).
Invalid property alternate ID	The property does not exist in the capability.
Illegal property value	The value is not acceptable for the reason specified in the same error message. The message can be used if the measure is posted as a list with a wrong number of elements. It can also be used when the value for a certain property cannot be converted to the corresponding type.
No valid measurement data found	No property of the capability was assigned a valid value.
Illegal time stamp value	The time stamp provided by the client for the measure message could not be parsed.
Gateway configuration error	Typically returned for a wrongly configured Internet of Things Gateway Edge. However, it may also be sent if there are errors in the data model, such as a capability without any properties.
Gateway internal error	Some other error was detected, and an internal consistency check has failed in the Internet of Things Gateway.

### 3.1.3 Command Message Format (JSON)

After a device has been successfully registered and a certain sensor has been assigned, the device can start receiving commands that conform to a capability defined for the respective sensor type.

The schema of a valid command JSON payload looks as follows:

```
{
  "title": "IoT Gateway JSON Message Schema",
  "type": "object",
```

```

"properties":{
  "sensorAlternateId":{
    "id":"sensorAlternateId",
    "type":"string",
    "required":true,
    "description":"alternate id of the sensor the measures belong to"
  },
  "capabilityAlternateId":{
    "id":"capabilityAlternateId",
    "type":"string",
    "required":true,
    "description":"alternate id of the capability the command conform to"
  },
  "command":{
    "id":"command",
    "type":"object",
    "description":"command that conform to the capability definition with
the given alternate id"
  }
}
}

```

The following is an example of a valid command message published by the Internet of Things Gateway. A device can consume the message, by subscribing to the commands topic as described in the section *Endpoint Structure*.

#### Sample Code

```

{
  "sensorAlternateId":"sensor2",
  "capabilityAlternateId":"capability2",
  "command":{
    "LED":"on",
    "Buzzer":"off",
    "Speed":45
  }
}

```

The payload includes a message that conforms to a capability with the alternate ID `capability2`. The command is for the sensor with the alternate ID `sensor2`. The command contains three properties, namely `LED`, `Buzzer`, and `Speed`.

## 3.1.4 Device Authorization and Authentication

The client authorization logic cross-checks the metadata provided by the connecting device through its client certificate with the resources the connecting device is trying to access.

In each client connection, the following authorization checks are performed:

- The `instanceId` included in the certificate must match the ID of the instance to which the Internet of Things Gateway belongs.
- The `deviceAlternateId` included in the certificate must match the ID of the instance to which the Internet of Things Gateway belongs.
- The `deviceAlternateId` included in the certificate must match the MQTT `clientId` retrieved from the MQTT metadata. This constraint can be relaxed in case the connecting device has the authorization type `router`, as described in section [Connection as a Router Device \[page 27\]](#).

If an MQTT client attempts to post measurement data for ingestion, the following authorization checks are performed:

- The topic to which the measurement data is published must match the structure that is defined for the data ingestion topic. For more information, please refer to section [Measure Message Format \(JSON\) \[page 13\]](#).
- The `deviceAlternateId` included in the certificate must match the `alternateId` value that is specified in the measure ingestion topic.

If an MQTT client attempts to subscribe to the command topic, the following authorization checks are performed:

- The target topic must match the structure that is defined for the command topic. For more information, please refer to section [Command Message Format \(JSON\) \[page 25\]](#).
- The `deviceAlternateId` included in the certificate must match the `alternateId` value that is specified in the command topic.

### 3.1.4.1 Connection as a Router Device

If the device has the authorization type `router`, it is allowed to send data and subscribe to the command topic on behalf of another device managed by the same gateway.

In this case, use the certificate of the router device. In the payload, use the device Alternate ID, capability Alternate ID, and sensor Alternate ID of the other device that you want to send measures to. For more information, please refer to the tutorial [Create a Device Model Using the API](#).

If the authorization type is `router`, the match between the `deviceAlternateId` and the MQTT `clientId` is not enforced. This allows a router device to connect using any value for the MQTT `clientId`.

### 3.1.4.2 Certificate Structure

Data ingestion on the Internet of Things Gateway Cloud endpoints uses TLS with mutual authentication by leveraging X.509 client certificates associated with single devices. The certificate field has the following structure:

```
CN=deviceAlternateId:<...>|gatewayId:<...>|tenantId:<...>|instanceId:<...>
```

The `deviceAlternateId` represents the alternate ID of the targeted device and the `instanceId` represents the ID associated with the Internet of Things Service instance that the Internet of Things Gateway Cloud belongs to.

For example, the subject of the certificate could look as follows:

#### Sample Code

```
CN=deviceAlternateId:1234562599|gatewayId:3|tenantId:1756591794|
instanceId:iot-certificate, OU=IoT Services, O=SAP Trust Community, L=, ST=,
C=DE
```

## 3.1.5 Endpoint Structure

The Port for the Internet of Things Gateway Cloud MQTT is 8883. Therefore, the server URI for the Internet of Things Gateway Cloud for MQTT is:

```
ssl://<HOST_NAME>:8883
```

A valid sample is:

### Sample Code

```
ssl://demo.eu10.cp.iot.sap:8883
```

Measures can be ingested into the Internet of Things Gateway Cloud MQTT by publishing data on a topic with the following structure:

```
measures/<deviceAlternateId>
```

The `deviceAlternateId` represents the alternate ID of the device that generated the measures.

Commands may be delivered to devices on the following topic:

```
commands/<deviceAlternateId>
```

The `deviceAlternateId` represents the alternate ID of the device to which commands are directed.

## 3.1.6 Command Management

Commands are implemented as follows:

- Devices subscribe to a command topic, whose structure is defined in section [Endpoint Structure \[page 28\]](#).
- The Internet of Things Gateway publishes command messages addressed to a device on the related command topic.

As soon as a command API (`/devices/{deviceId}/commands`) is invoked, a command message reaches the Internet of Things Gateway. In case an MQTT client is subscribed to the related command topic, it receives the message immediately. Also, the Internet of Things Gateway provides message persistency to enable the delivery of messages to unconnected MQTT clients as soon as they reestablish a connection.

### Delivery of Persisted Messages

To ensure the delivery of persisted messages, the MQTT client must perform the following operations:

- In the MQTT Connect packet: Connect to the Internet of Things Gateway with `cleanSession = false`.
- In the Subscribe packet: Subscribe to the command topic for each device of interest, using `QoS = 2` or `QoS = 1`.

After the first MQTT Connect/Subscribe operations have been performed on a command topic with the mentioned settings, the Internet of Things Gateway is able to deliver persisted commands addressed to the device. Before the first MQTT Connect/Subscribe, the delivery of persisted commands does not occur.

Command messages persist for up to three days after the command API is invoked.

## Delivery to Multiple Clients

The Internet of Things Gateway MQTT allows multiple MQTT clients to subscribe to the same command topic, if they are connected as a router device. For more information, please refer to section [Connection as a Router Device \[page 27\]](#).

In this case, the command message is delivered to every subscribed client.

## 3.2 REST

The Internet of Things Gateway Cloud for REST (HTTP) provides interfaces that are used by devices to send measures and receive commands. The following sections describe the message format that is provided by these interfaces for communication.

### Related Information

[Measure Message Format \(JSON\) \[page 29\]](#)

[Report the Measure Processing Results \(JSON\) \[page 39\]](#)

[Measure Message Format \(Protobuf\) \[page 42\]](#)

[Report the Measure Processing Results \(Protobuf\) \[page 44\]](#)

[Command Message Format \(JSON\) \[page 47\]](#)

[Command Message Format \(Protobuf\) \[page 48\]](#)

[Device Authorization and Authentication \[page 49\]](#)

[Endpoint Structure \[page 50\]](#)

### 3.2.1 Measure Message Format (JSON)

After a device has been successfully registered and a certain sensor has been assigned, the device can start sending messages that conform to a capability defined for the respective sensor type.

The schema of a valid measure message format looks as follows:

## Sample Code

```
{
  "oneOf": [
    {
      "$ref": "#/definitions/messageObject"
    },
    {
      "type": "array",
      "items": {
        "$ref": "#/definitions/messageObject"
      }
    }
  ],
  "definitions": {
    "messageObject": {
      "title": "IoT Gateway JSON Message Schema",
      "type": "object",
      "properties": {
        "sensorAlternateId": {
          "id": "sensorAlternateId",
          "type": "string",
          "required": true,
          "description": "alternate id of the sensor the measures belong
to"
        },
        "capabilityAlternateId": {
          "id": "capabilityAlternateId",
          "type": "string",
          "required": true,
          "description": "alternate id of the capability the measures
conform to"
        },
        "timestamp": {
          "oneOf": [
            {
              "id": "timestamp",
              "type": "string",
              "required": false,
              "description": "time of the measure retrieval at the
device in ISO 8601 format"
            },
            {
              "id": "timestamp",
              "type": "number",
              "required": false,
              "description": "time of the measure retrieval at the
device in Unix format in milliseconds since January, 1st 1970 "
            }
          ]
        },
        "measureMessageId": {
          "id": "measureMessageId",
          "type": "string",
          "required": false,
          "description": "an identifier of the message with measurement
data, unique within the device"
        },
        "processingTag": {
          "id": "processingTag",
          "type": "string",
          "required": false,
          "description": "property for dispatching measures to the
corresponding Message Selector"
        },
        "measures": {
          "id": "measures",
```

```

        "type": "array",
        "minItems": 1,
        "required": true,
        "items": {
          "oneOf": [
            {
              "type": "array",
              "minItems": 1,
              "required": true,
              "items": {
                "type": "string",
                "description": "list of measures that conform to
the capability definition with the given alternate id"
              }
            },
            {
              "type": "object",
              "minItems": 1,
              "required": true,
              "description": "list of measures that conform to the
capability definition with the given alternate id"
            }
          ]
        }
      }
    }
  }
}

```

### 3.2.1.1 Single Measures

The schema of a valid message with a single measure that is sent to the Internet of Things Gateway cloud looks as follows:

#### Sample Code

```

{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "measureMessageId": "123",
  "measures": [
    {
      "Humidity": 90,
      "Temperature": 23.5,
      "Status": "Good"
    }
  ]
}

```

The payload includes a message that conforms to a capability with the alternate ID `capability1`. The measures are for the sensor with the alternate ID `sensor1`. The measure contains three properties, namely `Humidity`, `Temperature`, and `Status`. These fields are part of the respective capability definition.

It is possible to send the measure properties in any order and leave out values that are unknown.

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "measures": [
    {
      "Status": "Good",
      "Humidity": 90
    }
  ]
}
```

## Timestamp

For each measure, the timestamp can be reported. The Internet of Things Service use timestamp to order the measurement samples returned by the Internet of Things Service API. If a timestamp is not specified in the ingestion payload, the time of arrival of the message will be referenced.

The timestamp can be provided in two ways:

- **Message Timestamp**

As described in the schema above, you can specify the ingestion time for the whole message through the property `timestamp`.

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "timestamp": 1413191650000,
  "measures": [
    {
      "Humidity": 90,
      "Temperature": 23.5
    }
  ]
}
```

Both formats, ISO 8601 and a Unix time in milliseconds from epoch (that is a 13-digit number), are allowed.

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "timestamp": "2017-05-30T15:06:15.123Z",
  "measures": [
    {
      "Humidity": 90,
      "Temperature": 23.5
    }
  ]
}
```



- **Measure Timestamp**

- If you have modeled a capability with a property of type `date`, you can use it to specify the measure timestamp (that is the timestamp of the sample).

#### Sample Code

```
{
  "sensorAlternateId": "sensor2",
  "capabilityAlternateId": "capability2",
  "measures": [
    {
      "Pressure": 90,
      "Temperature": 19.3,
      "Timestamp": 1413191650000
    }
  ]
}
```

Both formats, ISO 8601 and a Unix time in milliseconds from epoch (that is a 13-digit number), are allowed.

#### Sample Code

```
{
  "sensorAlternateId": "sensor2",
  "capabilityAlternateId": "capability2",
  "measures": [
    {
      "Pressure": 90,
      "Temperature": 19.3,
      "Timestamp": "2017-05-30T15:06:15.123Z"
    }
  ]
}
```

Looking at the example above, the payload includes a message that conforms to a capability with alternate ID `capability2`. The measures are for the sensor with the alternate ID `sensor2`. The measure contains three properties, namely `Pressure`, `Temperature`, and `Timestamp` (with the property `Timestamp` of type `date`). These fields are part of the respective capability definition. Modeling a property of type `date` is useful if:

- You want to use one of the properties of the device as timestamp of the measure.
- You want to send a single message with batched measures, for example, a message containing more samples for the capability, each one with a different timestamp (as for a default time series).  
For more information, please refer to section [Batched Measures \[page 19\]](#).
- You can use a special property named `'_time'` and not modelled into the capability to specify the measure timestamp (that is the timestamp of the sample).

#### Sample Code

```
{
  "sensorAlternateId": "sensor3",
  "capabilityAlternateId": "capability3",
  "measures": [
    {
      "Pressure": 90,
      "Temperature": 19.3,
      "_time": 1413191650000
    }
  ]
}
```

```
    ]  
  }
```

Both formats, ISO 8601 and a Unix time in milliseconds from epoch (that is a 13-digit number), are allowed.

### Sample Code

```
{  
  "sensorAlternateId": "sensor3",  
  "capabilityAlternateId": "capability3",  
  "measures": [  
    {  
      "Pressure": 90,  
      "Temperature": 19.3,  
      "_time": "2017-05-30T15:06:15.123Z"  
    }  
  ]  
}
```

Looking at the example above, the payload includes a message that conforms to a capability with alternate ID `capability3`. The measures are for the sensor with the alternate ID `sensor3`. The measure contains three properties, namely `Pressure`, `Temperature`, and `_time`, but only `Pressure` and `Temperature` are part of the respective capability definition.

Using the special `_time` property is useful if:

- You want to use it as timestamp of the measure, without modelling it as a property of the capability.
- You want to send a single message with batched measures, for example, a message containing more samples for the capability, each one with a different timestamp (as for a default time series). For more information, please refer to section [Batched Measures \[page 19\]](#).

### Note

- Both, the Message Timestamp and the Measure Timestamp are normalized to Unix time in milliseconds from epoch (that is a 13-digit number) after the ingestion and represented as long.
- The Measure Timestamp has a higher priority than the Message Timestamp, for example, if both are present, the platform uses the Measure Timestamp as ingestion timestamp.
- The Measure Timestamp expressed via the special property `"_time"` has a higher priority to the Measure Timestamp expressed via a property of type `'date'`; for example, if both are present, the platform uses the value of the `_time` property as ingestion timestamp.

## Processing Tag

A message can be tagged to be dispatched to the desired message selector. To do this, after have configured the processing of messages through the Message Processing API, you can specify the `processingTag` property in the message. Further information about the Internet of Things Message Processing, selectors, and their configuration can be found in section in the *Internet of Things Message Processing* documentation.

The following sample payload relies on a message selector using the `processingTag` property with value `sampleTag`.

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "processingTag": "sampleTag",
  "measures": [
    {
      "Status": "Good",
      "Humidity": 90
    }
  ]
}
```

## 3.2.1.2 Batched Measures

It is possible to send multiple measures that conform to the same capability in a single message.

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "measures": [
    {
      "Humidity": 90,
      "Temperature": 23.5,
      "Status": "Good"
    },
    {
      "Humidity": 93,
      "Temperature": 24.0,
      "Status": "Bad"
    }
  ]
}
```

## Timestamp

- Message Timestamp

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "timestamp": 1413191650000,
  "measures": [
    {
      "Humidity": 90,
      "Temperature": 23.5,
      "Status": "Good"
    }
  ],
}
```

```

    {
      "Humidity":93,
      "Temperature":24.0,
      "Status":"Bad"
    }
  ]
}

```

If you specify the timestamp as message timestamp, it will be applied to all the samples (for example, in this case both, the first and the second samples, will have the same ingestion timestamp 1413191650000)

- **Measure Timestamp**

- If you have modeled a property as type `date`, its value will result in the ingestion timestamp of each sample on the platform.

#### ↔ Sample Code

```

{
  "sensorAlternateId":"sensor2",
  "capabilityAlternateId":"capability2",
  "measures":[
    {
      "Pressure":90,
      "Temperature":19.3,
      "Timestamp": 1413191650000
    },
    {
      "Pressure":75,
      "Temperature":10.1,
      "Timestamp": 1413191790000
    }
  ]
}

```

For example, in this case you have two samples, the first one with 1413191650000 as ingestion timestamp and the second one with 1413191790000.

- If you have specified a `_time` property, without having modeled it as a property of the capability, its value will result in the ingestion timestamp of each sample on the platform.

#### ↔ Sample Code

```

{
  "sensorAlternateId":"sensor3",
  "capabilityAlternateId":"capability3",
  "measures":[
    {
      "Pressure":90,
      "Temperature":19.3,
      "_time": 1413191650000
    },
    {
      "Pressure":75,
      "Temperature":10.1,
      "_time": 1413191790000
    }
  ]
}

```

For example, in this case you have two samples, the first one with 1413191650000 as ingestion timestamp and the second one with 1413191790000.

### 3.2.1.3 Compressed Single Measures

It is possible to compress the measures by omitting the field names. In this case, the measures array is an array of arrays and the order must match the order of the properties in the capability definition.

#### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "measures": [
    [
      90,
      23.5,
      "Good"
    ]
  ]
}
```

It is also possible to send null values for properties for which no values exist.

#### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "measures": [
    [
      90,
      null,
      "Good"
    ]
  ]
}
```

### 3.2.1.4 Compressed Batched Measures

It is possible to send multiple measures with the compressed format (array) in a single message.

#### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "measures": [
    [
      90,
      23.5,
      "Good"
    ],
    [
      94,
      24.0,
      "Bad"
    ]
  ]
}
```

```
]
```

### 3.2.1.5 Batched Measure Messages

It is possible to send a batch of messages that conform to different capabilities. In this case, the batch contains an array of messages.

#### Sample Code

```
[
  {
    "sensorAlternateId": "sensor1",
    "capabilityAlternateId": "capability1",
    "measures": [
      {
        "Humidity": 90,
        "Temperature": 23.5,
        "Status": "Good"
      },
      {
        "Humidity": 93,
        "Temperature": 24.0,
        "Status": "Bad"
      }
    ]
  },
  {
    "sensorAlternateId": "sensor2",
    "capabilityAlternateId": "capability2",
    "measures": [
      {
        "Pressure": 90,
        "Temperature": 19.3,
        "Timestamp": 1413191650000
      }
    ]
  }
]
```

### 3.2.1.6 Measures for the Auto-Onboarding of Sensors

It is possible to auto-onboard sensors during the data ingestion. Auto-onboarding means that sensors that do not exist are created automatically during data ingestion. In this case, an additional attribute, namely the `sensorTypeAlternateId`, must be passed. If the sensor with the given `sensorAlternateId` does not exist, a new sensor is created for the device that is assigned to the sensor type with this `sensorTypeAlternateId`.

#### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "sensorTypeAlternateId": "12",
  "capabilityAlternateId": "capability1",
```

```
"measures": [
  {
    "Humidity": 90,
    "Temperature": 23.5,
    "Status": "Good"
  }
]
```

#### Note

Only numeric values are supported for the alternate ID of a sensor type.

#### Note

The following error codes are returned for failing auto-onboarding attempts against the Internet of Things Gateway Cloud REST:

- 401 if the device does not exist.
- 400 if the message format is not correct, for example, one of the `sensorAlternateId`, `sensorTypeAlternateId`, or `capabilityAlternateId` is missing.

## 3.2.2 Report the Measure Processing Results (JSON)

The Internet of Things Gateway reports the processing results for the posted measures to the device using the REST API. The general outcome is reported as the status code returned for the corresponding HTTP request. The details including the error messages are returned in the body of the HTTP response which has the same content type as the request: if the measures are posted in the JSON format, the detailed processing results are also returned in JSON. If the measures are posted in the Google Protocol Buffers format, the results are returned in protobuf. For more information, please refer to section [Report the Measure Processing Results \(Protobuf\)](#) [page 44].

Note that the Internet of Things Gateway reports the processing status immediately, before forwarding the measures to the Internet of Things Service. If the Internet of Things Gateway considers the measure message invalid and rejects it, this outcome is final. However, if the Internet of Things Gateway accepts the measure message as valid, it may still be rejected or lost further down the road. Thus, the Internet of Things Gateway typically does not return the 200 OK status code, but only 202 Accepted. The REST API will be extended in future releases of the gateway, allowing HTTP clients to query the final processing outcome of measure messages posted earlier, including the processing by the Internet of Things Service.

### 3.2.2.1 HTTP Status Codes

The device must be able to handle at least the status codes described in the following table. This list is not exhaustive. Future versions of the Internet of Things Gateway may return additional status codes.

Status code	Meaning
200 OK	The Internet of Things Gateway has received a confirmation from the Internet of Things Service that the measure message was processed correctly before returning the result to the device. Almost never occurs.
202 Accepted	The measure message has passed the validation check executed by the Internet of Things Gateway.
207 Multi-Status	<a href="#">Batched Measure Messages [page 22]</a> were posted to the Internet of Things Gateway. Some messages in the batch have failed the validation, but at least one was accepted. The body of the response contains detailed information.
400 Bad Request	The measure message has failed the validation check executed by the Internet of Things Gateway. If the message posted is a batched message, all message parts have failed the validation check. All measures were discarded. The body of the response contains detailed information.
401 Unauthorized	Authentication has failed, or the authenticated device is not authorized to post measures on behalf of the device with the alternate ID specified in the URL. For more information, please refer to section <a href="#">Device Authorization and Authentication [page 49]</a> .
404 Not Found	The requested URL refers to a non-existent REST endpoint.
405 Method Not Allowed	Only HTTP POST can be used for sending the measures.
500 Internal Server Error	The Internet of Things Gateway is not configured correctly, or a software error has occurred in the Internet of Things Gateway.
504 Gateway Timeout	This error is not returned by the Internet of Things Gateway itself, but rather by an HTTP proxy or reverse proxy located between the device and the Internet of Things Gateway. The most likely reason is that the Internet of Things Gateway is overloaded with requests, or that the measure message posted is very large and takes too long to process.

### 3.2.2.2 Response Body

When the device posts measures in the JSON format, it receives a detailed status response which is also JSON-formatted. The device must check the Content-Type HTTP header because a different content type may be returned by an HTTP proxy, or no response body at all may be returned in some cases.

The detailed status message is a list of objects. The number of objects is either one or corresponds to the number of measure messages posted within [Batched Measure Messages \[page 22\]](#). In the latter case, the



order of the objects is the same as the order of the measure messages in the batch. Each object can have the following fields:

```
{
  "code": "the status code for this Measure Message; the same as the HTTP status
  code if there is only one Measure Message in the HTTP request",
  "sensorAlternateId": "value from the request",
  "capabilityAlternateId": "value from the request",
  "messages": [a list of human-readable strings, mostly error messages, generated
  during the verification of the measure message]
}
```

Subsequent versions of the Internet of Things Gateway may extend the object with additional fields. All fields except for the `code` field can be missing (for example, if the JSON request could not be parsed, or no errors were detected).

The semantics of the `code` field is the same as of the corresponding HTTP status code (see the following table).

The `messages` field can contain any human-readable error messages. There is no fixed list of allowed values, but the messages may start with the strings listed in the table below.

Typical error messages are described in the following table.

Message	Condition
Unauthorized	The client is not authorized to post the measure messages on behalf of the device. In particular, this may mean an HTTP authentication failure.
Invalid device alternate ID	Most likely to appear when a valid and authenticated router device posts measure messages on behalf of another one.
Invalid message format	This means that the whole payload could not be parsed, for example due to invalid JSON syntax. The sensor and device alternate IDs, etc., cannot be extracted in this case.
Invalid sensor alternate ID	The sensor does not exist and could not be onboarded automatically.
Invalid capability alternate ID	The capability does not exist for the sensor, or it does not accept measures (being of the command type).
Invalid property alternate ID	The property does not exist in the capability.
Illegal property value	The value is not acceptable for the reason specified in the same error message. The message can be used if the measure is posted as a list with a wrong number of elements. It can also be used when the value for a certain property cannot be converted to the corresponding type.
No valid measurement data found	No property of the capability was assigned a valid value.

Message	Condition
Illegal time stamp value	The time stamp provided by the client for the measure message could not be parsed.
Gateway configuration error	Typically returned for a wrongly configured Internet of Things Gateway Edge. However, it may also be sent if there are errors in the data model, such as a capability without any properties.
Gateway internal error	Some other error was detected, and an internal consistency check has failed in the Internet of Things Gateway.

### 3.2.3 Measure Message Format (Protobuf)

You can use the Internet of Things Gateway REST to ingest measures into the system in a binary format by using Google Protocol Buffers (protobuf).

The ingestion is made in the same way and at the same address as for the JSON format. To specify the use of the protobuf format in the POST request, set the value of the `Content-Type` attribute to `application/x-protobuf`:

```
Content-Type:application/x-protobuf
```

The body should be a binary (bytes) representation of a protobuf message that is compliant with the format declared by the file `MeasureRequest.proto`, as follows:

```
syntax = "proto3";
import "google/protobuf/any.proto";
package gateway;
option java_package = "com.sap.iotservices.common.protobuf.gateway";
option java_outer_classname = "MeasureProtos";
message MeasureRequest {
  string capabilityAlternateId = 1;
  string sensorAlternateId = 2;
  string sensorTypeAlternateId = 3;
  int64 timestamp = 4;
  repeated Measure measures = 5;

  message Measure {
    repeated google.protobuf.Any values = 1;
  }
}
```

When you create a protocol buffers schema for an existing message type consider the following:

- Protocol buffers do not support field names that start with numeric values and contain non-alphanumeric characters (except for the character '\_'). Make sure that you name your message type fields accordingly.
- The protocol buffers field names, types, and positions must match the message type. The position can be different, but in that case the measures field must be built in a specific way. For more information, please refer to section [Measure Value \[page 43\]](#).
- Fields of type `google.protobuf.Any` are stored as binary fields. We recommend that you use flat data structures.

### 3.2.3.1 Measure Identifier

- `Field_id`: `capabilityAlternateId`
- Field format: string

This field represents an identifier for the measure that is referred to by a measurement report. It maps to the capability alternate ID, which univocally identifies a certain measurement source hosted on the node. The field has the form of a single string, if the payload reports a single measurement.

### 3.2.3.2 Measure Timestamp

- `Field_id`: `timestamp`
- Field format: Java timestamp (Unix timestamp + milliseconds, for example, 1447407509060).

This field reports the timestamp of the measurement collection time, that is, the time of generation of the measurement on the device. This value is referred to as the measurement `start_time`, while the time of arrival in the Internet of Things Gateway is also known as `arrival_time`. This field is optional. If omitted, the Internet of Things Gateway parsing logic sets the measurement `start_time` equal to the `arrival_time`.

### 3.2.3.3 Measure Value

- `Field_id`: `measures`
- Field format: sequence (array) of sequences of the `google.protobuf.Any` type

This field reports the measurement values. The field can have the form of

- an array with only one inner array with a value for each inner property, in case the payload reports a single measure
- An array with more inner arrays, each of which with a value for each inner property, in case the payload reports a multiple measure.

#### Note

In this case, one of the properties of the capability must be of the type `measureTimestamp` and contain a different timestamp (expressed as milliseconds from epoch) for each measure.

To represent a capability with multiple properties, they are included as several `google.protobuf.Any` values in the same measure object, which results in a `measures` field that consists of only one value of the type `Measure`. This field contains several `values` fields, one for each property to represent.

It is allowed to have a different field order within the `Any` value than the one defined in the capability. In this case, the `typeName` value of the `Any` type is used to recognize the correct field type. Therefore, be sure that `typeName` contains the name of the property as defined in the capability.

If `typeName` is empty, the same order as defined in the capability is used to resolve the fields.

On the other hand, several `measures` objects of the type `Measure` are used to represent a time series. Each of these `measures` objects contains a single `values` field with one value of the series.

### 3.2.3.4 Sensor Alternate ID

- `Field_id`: `sensorAlternateId`
- Field format: ASCII string

This field reports the alternate ID of the sensor to which the measurement refers. If omitted, the Internet of Things Gateway automatically associates the measurement to a default sensor with the alternate ID `00:00:00:01`.

### 3.2.3.5 SensorType Alternate ID

- `Field_id`: `sensorTypeAlternateId`
- Required: **NO**
- Field format: integer

This field reports the `sensorTypeAlternateId` to which the measurement refers. If omitted, the gateway automatically associates the measurement to the default `sensorType` for the specific protocol (for example, 0 for CoAP, MQTT, REST). The field must always be in the form of a single, scalar number. Thus, the related `sensorTypeAlternateId` applies to the entire set of measurements reported in the current payload.

## 3.2.4 Report the Measure Processing Results (Protobuf)

The Internet of Things Gateway reports the processing results for the posted `measures` to the device using the REST API. The general outcome is reported as the status code returned for the corresponding HTTP request. The details including the error messages are returned in the body of the HTTP response which has the same content type as the request: if the `measures` are posted in the Google Protocol Buffers format, the detailed processing results are also returned in protobuf. If the `measures` are posted in the JSON format, the results are returned in JSON. For more information, please refer to section [Report the Measure Processing Results \(JSON\) \[page 39\]](#).

Note that the Internet of Things Gateway reports the processing status immediately, before forwarding the `measures` to the Internet of Things Service. If the Internet of Things Gateway considers the measure message invalid and rejects it, this outcome is final. However, if the Internet of Things Gateway accepts the measure message as valid, it may still be rejected or lost further down the road. Thus, the Internet of Things Gateway typically does not return the `200 OK` status code, but only `202 Accepted`. The REST API will be extended in future releases of the gateway, allowing HTTP clients to query the final processing outcome of measure messages posted earlier, including the processing by the Internet of Things Service.

## 3.2.4.1 HTTP Status Codes

The device must be able to handle at least the status codes described in the following table. This list is not exhaustive. Future versions of the Internet of Things Gateway may return additional status codes.

Status code	Meaning
200 OK	The Internet of Things Gateway has received a confirmation from the Internet of Things Service that the measure message was processed correctly before returning the result to the device. Almost never occurs.
202 Accepted	The measure message has passed the validation check executed by the Internet of Things Gateway.
207 Multi-Status	<a href="#">Batched Measure Messages [page 22]</a> were posted to the Internet of Things Gateway. Some messages in the batch have failed the validation, but at least one was accepted. The body of the response contains detailed information.
400 Bad Request	The measure message has failed the validation check executed by the Internet of Things Gateway. If the message posted is a batched message, all message parts have failed the validation check. All measures were discarded. The body of the response contains detailed information.
401 Unauthorized	Authentication has failed, or the authenticated device is not authorized to post measures on behalf of the device with the alternate ID specified in the URL. For more information, please refer to section <a href="#">Device Authorization and Authentication [page 49]</a> .
404 Not Found	The requested URL refers to a non-existent REST endpoint.
405 Method Not Allowed	Only HTTP POST can be used for sending the measures.
500 Internal Server Error	The Internet of Things Gateway is not configured correctly, or a software error has occurred in the Internet of Things Gateway.
504 Gateway Timeout	This error is not returned by the Internet of Things Gateway itself, but rather by an HTTP proxy or reverse proxy located between the device and the Internet of Things Gateway. The most likely reason is that the Internet of Things Gateway is overloaded with requests, or that the measure message posted is very large and takes too long to process.

## 3.2.4.2 Response Body

When the device posts measures in the protobuf format, it receives a detailed status response which is also in the protobuf format. The device must check the Content-Type HTTP header because a different content type may be returned by an HTTP proxy, or no response body at all may be returned in some cases.

The definition of the corresponding type in the proto3 syntax is provided in the following sample. All fields are optional. More fields can be added in future versions of the Internet of Things Gateway.

```
message MeasureResponse {
  int32 code = 1; // the same as the HTTP status code
  string id = 2; // not returned by the current version
  int64 timestamp = 3; // not returned by the current version
  string sensorAlternateId = 4; // value from the request
  string capabilityAlternateId = 5; // value from the request
  repeated string messages = 6; // a list of human-readable strings,
                                // mostly error messages, generated during
                                // the verification of the measure message
}
```

The current version of the Internet of Things Gateway does not support the protobuf format for batched measure messages. Therefore, neither individual identifiers nor individual timestamps for measure messages are needed.

The `messages` field can contain any human-readable error messages. There is no fixed list of allowed values, but the messages may start with the strings listed in the following table.

Typical error messages are described in the following table.

Message	Condition
Unauthorized	The client is not authorized to post the measure messages on behalf of the device. In particular, this may mean an HTTP authentication failure.
Invalid device alternate ID	Most likely to appear when a valid and authenticated router device posts measure messages on behalf of another one.
Invalid message format	This means that the whole payload could not be parsed, for example due to invalid JSON syntax. The sensor and device alternate IDs, etc., cannot be extracted in this case.
Invalid sensor alternate ID	The sensor does not exist and could not be onboarded automatically.
Invalid capability alternate ID	The capability does not exist for the sensor, or it does not accept measures (being of the command type).
Invalid property alternate ID	The property does not exist in the capability.

Message	Condition
Illegal property value	The value is not acceptable for the reason specified in the same error message. The message can be used if the measure is posted as a list with a wrong number of elements. It can also be used when the value for a certain property cannot be converted to the corresponding type.
No valid measurement data found	No property of the capability was assigned a valid value.
Illegal time stamp value	The time stamp provided by the client for the measure message could not be parsed.
Gateway configuration error	Typically returned for a wrongly configured Internet of Things Gateway Edge. However, it may also be sent if there are errors in the data model, such as a capability without any properties.
Gateway internal error	Some other error was detected, and an internal consistency check has failed in the Internet of Things Gateway.

### 3.2.5 Command Message Format (JSON)

After a device has been successfully registered and a certain sensor has been assigned, the device can start receiving commands that conform to a capability defined for the respective sensor type.

The schema of a valid command JSON payload looks as follows:

```
{
  "title": "IoT Gateway JSON Message Schema",
  "type": "object",
  "properties": {
    "sensorAlternateId": {
      "id": "sensorAlternateId",
      "type": "string",
      "required": true,
      "description": "alternate id of the sensor the measures belong to"
    },
    "capabilityAlternateId": {
      "id": "capabilityAlternateId",
      "type": "string",
      "required": true,
      "description": "alternate id of the capability the command conform to"
    },
    "command": {
      "id": "command",
      "type": "object",
      "description": "command that conform to the capability definition with the given alternate id"
    }
  }
}
```

The following is an example of a valid command message published by the Internet of Things Gateway. A device can consume the message, by subscribing to the commands topic as described in the section *Endpoint Structure*.

#### Sample Code

```
{
  "sensorAlternateId": "sensor2",
  "capabilityAlternateId": "capability2",
  "command": {
    "LED": "on",
    "Buzzer": "off",
    "Speed": 45
  }
}
```

The payload includes a message that conforms to a capability with the alternate ID `capability2`. The command is for the sensor with the alternate ID `sensor2`. The command contains three properties, namely `LED`, `Buzzer`, and `Speed`.

## 3.2.6 Command Message Format (Protobuf)

After a device has been successfully registered and a certain sensor has been assigned, the device can start receiving commands that conform to a capability defined for the respective sensor type.

It is possible to send commands as binary data by using Google Protocol Buffers (protobuf) encoding as payload for command messages. The format is shown in the following `CommandResponse.proto` file:

```
syntax = "proto3";
import "google/protobuf/any.proto";
package gateway;
option java_package = "com.sap.iotservices.common.protobuf.gateway";
option java_outer_classname = "CommandResponseProtos";
message CommandResponse {
  string capabilityAlternateId = 1;
  string sensorAlternateId = 2;
  Command command = 3;

  message Command {
    repeated google.protobuf.Any values = 1;
  }
}
```

The `command` field is of type `Command`, which is defined as a sequence (array) of type `google.protobuf.Any` with the purpose to match the properties of the capability that represents the command specified with the capability alternate ID.

In case multiple commands are delivered to the device, they are included in a `CommandList` message, so that they are delivered as an array of command objects. The following `CommandResponseList.proto` file contains the syntax:

```
syntax = "proto3";
import "CommandResponse.proto";
package gateway;
option java_package = "com.sap.iotservices.common.protobuf.gateway";
option java_outer_classname = "CommandResponseListProtos";
```



```
message CommandResponseList {
  repeated CommandResponse commands = 1;
}
```

When you create a protocol buffers schema for an existing message type consider the following:

- Protocol buffers do not support field names that start with numeric values and contain non-alphanumeric characters (except for the character '\_'). Make sure that you name your message type fields accordingly.

## 3.2.7 Device Authorization and Authentication

The client authorization logic cross-checks the metadata provided by the connecting device through its client certificate with the resources the connecting device is trying to access.

In each client connection, the following authorization checks are performed:

- The instance ID included in the certificate must match the ID of the instance to which the Internet of Things Gateway belongs.
- The device alternate ID included in the certificate must match the device alternate ID used in the URL path.

When a REST client performs a GET on the command endpoint, the following authorization checks are performed:

- The target URL must match the structure defined for the command endpoint.
- The instance ID included in the certificate must match the ID of the instance to which the Internet of Things Gateway belongs.
- The device alternate ID included in the certificate must match the alternate ID value specified in the command endpoint URL.

### 3.2.7.1 Connection as a Router Device

If a device has the router authorization type, it is allowed to send data and retrieve commands on behalf of another device managed by the same gateway.

In this case, use the certificate of the router device. In the payload, use the device Alternate ID, capability Alternate ID, and sensor Alternate ID of the other device that you want to send measures to. For more information, please refer to the tutorial [Create a Device Model Using the API](#).

### 3.2.7.2 Certificate Structure

Data ingestion on the Internet of Things Gateway Cloud endpoints uses TLS with mutual authentication by leveraging X.509 client certificates associated with single devices. The certificate field has the following structure:

```
CN=deviceAlternateId:<...>|gatewayId:<...>|tenantId:<...>|instanceId:<...>
```

The `deviceAlternateId` represents the alternate ID of the targeted device and the `instanceId` represents the ID associated with the Internet of Things Service instance that the Internet of Things Gateway Cloud belongs to.

For example, the subject of the certificate could look as follows:

#### Sample Code

```
CN=deviceAlternateId:1234562599|gatewayId:3|tenantId:1756591794|
instanceId:iot-certificate, OU=IoT Services, O=SAP Trust Community, L=, ST=,
C=DE
```

### 3.2.7.3 Device Connection

In each client connection, the following authorization checks are performed:

- The instance ID included in the certificate must match the instance ID of the instance to which the Internet of Things Gateway Cloud belongs.
- The device alternate ID included in the certificate must match the device alternate ID in the URL path.

### 3.2.8 Endpoint Structure

Measures can be ingested into the Internet of Things Gateway Cloud REST by performing a POST call on the URL:

```
https://<HOST_NAME>:443/iot/gateway/rest/measures/<deviceAlternateId>
```

The `deviceAlternateId` represents the alternate ID of the device that generated the measures.

Commands can be retrieved from the Internet of Things Gateway by performing a GET call on the following URL:

```
https://<HOST_NAME>:443/iot/gateway/rest/commands/<deviceAlternateId>
```

# 4 Internet of Things Edge Platform

Please refer to the tutorial for the download and set up.

## Related Information

[Network Requirements \[page 51\]](#)

[OS Requirements \[page 52\]](#)

[Audit Log \[page 53\]](#)

[Offline Operations \[page 54\]](#)

[MQTT \[page 59\]](#)

[REST \[page 81\]](#)

[CoAP \[page 111\]](#)

[FILE \[page 114\]](#)

[Modbus \[page 115\]](#)

[OPC UA \[page 121\]](#)

[SigFox \[page 133\]](#)

[SNMP \[page 143\]](#)

## 4.1 Network Requirements

The communication between the Internet of Things Edge Platform and the Internet of Things Service relies on the following channels:

- HTTPS (port 443), for the consumption of Internet of Things Service APIs
- JMS (port 61616), for the dispatch of JMS frames on a secure NIO+SSL channel

When the Internet of Things Edge Platform runs on a private network, a proxy may be used for the HTTPS flow. The proxy configuration relies on the following Java VM arguments:

```
https.proxyHost=<proxy_host>
https.proxyPort=<proxy_port>
https.proxyUser=<username>
https.proxyPassword=<password>
java.net.useSystemProxies=true
```

Parameters `https.proxyUser` and `https.proxyPassword` may be omitted if the proxy does not require authentication.

You can pass the parameters specified to the Internet of Things Edge Platform by adding them to the startup script located in the Internet of Things Edge Platform root folder. The file name is `gateway.sh` for Unix and `gateway.bat` for Windows.

- Open the startup script.
- Add the parameters and the correct values. Make sure that the parameter names have the `-D` prefix and the parameter lines end with `\` or `^`, just like the other lines in the file. Make sure to insert the required lines before the last script line starting with the `-jar` command. The configuration in the `gateway.sh` file must be as follows:

```
-Dhttps.proxyHost=<proxy_host> \  
-Dhttps.proxyPort=<proxy_port> \  
-Djava.net.useSystemProxies=true \  
-Dhttps.proxyUser=<username> \  
-Dhttps.proxyPassword=<password> \  

```

- Save and close the startup script.

As the JMS flow cannot be routed through a standard HTTP proxy, an outgoing connection must be allowed from the network where the Internet of Things Edge Platform runs to port 61616 of the cloud instance it connects to.

## 4.2 OS Requirements

The target system where the Internet of Things Edge Platform is to be installed needs to fulfill the requirements at OS level described in this section.

### Java Support

Java SE Runtime Environment 8 must be installed on the target operating system, the minimum required version is 1.8.0\_192.

### Random Number Generation

On Unix platforms, the Java library used for random number generation typically relies on the random bytes provided by the device file `/dev/random`. In some cases, `/dev/random` data generation may be very slow, affecting negatively or even blocking any JVM call that relies on it (including, for example, the initialization of security libraries).

To determine whether a system has a slow `/dev/random` source, try to display a portion of the file with the following command:

```
head -n 1 /dev/random.
```

If the command does not return a result immediately, this is usually an indicator of a slow `/dev/random`. In this case, try the following workarounds:

- **Replacing the random source:** Although `/dev/random` is more secure, it is possible to use `/dev/urandom` as an alternative source which is guaranteed not to block. It is possible to enforce this setting by:
  - Adding a system property named `java.security.egd` to the `gateway.sh` or `gateway.bat` file:  
`-Djava.security.egd=file:/dev/./urandom`
  - Changing the `java.security` file under `$JAVA_HOME/jre/lib/security`:  
the line `securerandom.source=file:/dev/random` must be changed to  
`securerandom.source=file:/dev/./urandom`.  
Please take into account that this setting is applied JVM-wide, hence it affects other applications running on the same JVM. For more information, please refer to section [Avoiding JVM Delays Caused by Random Number Generation](#) in the *ORACLE™ Configuration Guide*.
- **Improving `/dev/random` generation:** The `/dev/random` generation can be accelerated by improving the system entropy. The [haveged daemon](#) aims at providing an unpredictable random number generator based on an adaptation of the HAVEGE algorithm. It was created to remedy low-entropy conditions in the Linux random device.

To install the daemon, follow these steps:

- **On Ubuntu/Debian:**
  - `apt-get install haveged`
  - `update-rc.d haveged defaults`
  - `service haveged start`
- **On RHEL/CentOS:**
  - `yum install haveged`
  - `systemctl enable haveged`
  - `systemctl start haveged`

## 4.3 Audit Log

The Internet of Things Edge Platform MQTT and the Internet of Things Edge Platform REST support the audit logging of the security events occurring on them.

In particular:

- The Internet of Things Edge Platform MQTT logs the occurrences of the following security events:
  - MQTT certificate-based connections
  - MQTT certificate-based publishing
  - MQTT certificate-based subscriptions
- The Internet of Things Edge Platform REST logs the occurrences of the following security events:
  - HTTPS connections requests

Any security event related to the previous conditions is logged by the platform into specific `.log` files, placed under the folder `/log` and the name starts with the pattern `'AuditLog'`.

It is possible to customize the default `log4j` configuration file (changing, for example, the name of the log file, its size, or modifying the rollover strategy or, again, the log level or the pattern layout) by editing the file `'log4j2.xml'` under the folder `'/config'`.

To do this, you can edit the `'log4j2.xml'` section

#### Sample Code

```
<RollingFile name="fileLogger" fileName="./log/AuditLog.log"
filePattern="./log/AuditLog-%i.log">
  <PatternLayout>
    <pattern>%d{ddMMM HH:mm:ss,SSS} [%t] %-5p (%c{1}:%L) - %m%n</
pattern>
  </PatternLayout>
  <Policies>
    <SizeBasedTriggeringPolicy size="10 MB" />
  </Policies>
  <DefaultRolloverStrategy max="10" />
</RollingFile>
```

to customize the settings about the `log4j` rolling file, like the name of the log files, the pattern layout, the default triggering policy or the default rollover strategy;

and/or the section

#### Sample Code

```
<Logger name="com.sap.iotservices.common.auditlog.service.impl" level="WARN"
additivity="false">
  <AppenderRef ref="fileLogger"/>
</Logger>
```

to customize the behavior of the logger used by the Edge Platform and change, for example, the default log level or the additivity configuration.

## 4.4 Offline Operations

The term **Offline Operations** refers to a set of operations that empower the Internet of Things Edge Platform administrator to fully control the Edge Platform synchronization with the Cloud, thereby making it possible to manually block or trigger it based on some business or operations criteria.

These offline operations support the following functionalities:

- The possibility to enable or disable data transfer to the Cloud
- The possibility to enable or disable the consumption of commands from the Cloud
- The possibility to enable or disable the device model synchronization with the Cloud
- The possibility to enable or disable the data model synchronization with the Cloud

There are two configuration files available to support the configuration of the required service. These files are present, by default, under the folder `/config/services` at the root of the Edge Gateway.

The default values are configured to leave the Gateway behavior unchanged with respect to the older versions; Netty Server is disabled and all the channels are enabled, that is, the communication with the Cloud is fully active.

The two configuration files are:

- `sap.NettyConfiguration.cfg`: This file contains the configurations to enable a local Netty Server that is used to allow calling local APIs.

**Default content:**

```
# Enables a Netty server
enabled = false
# Host name or IP address to identify a specific network interface on
# which to listen
host = localhost
# TCP/IP port on which the server listens for connections
port = 8900
```

- `sap.RemotingStrategyConfiguration.cfg`: This file contains the starting configuration for all the four operations.

**Default content:**

```
# Enable/Disable channel to send data to the Internet of Things
# Service Cloud
dataChannelActive = true
# Enable/Disable channel to send/receive device model updates from
# the Internet of Things Service Cloud
deviceModelChannelActive = true
# Enable/Disable channel to receive commands from the Internet of
# Things Service Cloud
commandsChannelActive = true
# Enable/Disable channel to receive data model updates from the
# Internet of Things Service Cloud
dataModelChannelActive = true
```

## Related Information

[Configuration \[page 55\]](#)

[Access using Swagger \[page 57\]](#)

[Buffering on Internet of Things Edge Platform \[page 57\]](#)

### 4.4.1 Configuration

To configure the single channels at startup, you need to edit the file `sap.RemotingStrategyConfiguration.cfg`, set `false` as the value for the channel(s) to stop. If you have to change the values at runtime, you also need to edit the file `sap.NettyConfiguration.cfg`, set `true` as the value of the field `enabled` and change, if needed, the port used. In this way, the Netty Server will run, allowing to call the local APIs that can modify channel activation or deactivation at runtime.

All those APIs can be found under the path `admin/sync` after the Edge API base path. They are available at the path:

```
iot/edge/api/v1/admin/sync
```

## Related Information

[Enable or Disable Data Transfer to the Cloud \[page 56\]](#)

[Enable or Disable Consumption of Commands from the Cloud \[page 56\]](#)

[Enable or Disable Device Model Synchronization with the Cloud \[page 57\]](#)

[Enable or Disable Data Model Synchronization with the Cloud \[page 57\]](#)

### 4.4.1.1 Enable or Disable Data Transfer to the Cloud

You can expose a local REST API to set or rest a flag that controls the forwarding of ingestion data to the Cloud.

- API name: `dataChannelActive`
- Body parameter: `active` (boolean)
- Example:

```
curl -X POST "http://<hostName>:<port>/<instanceId>/iot/edge/api/v1/admin/sync/  
dataChannelActive" -H "accept: */*" -H "Content-Type: application/json" -d  
"{\"active\": true}"
```

### 4.4.1.2 Enable or Disable Consumption of Commands from the Cloud

You can expose a local REST API to enable or disable command consumption. The API controls both the consumptions through SoapOverJMS queue and virtual topic. No differentiation between the two is exposed.

- API name: `commandsChannelActive`
- Body parameter: `active` (boolean)
- Example:

```
curl -X POST "http://<hostName>:<port>/<instanceId>/iot/edge/api/v1/admin/sync/  
commandsChannelActive" -H "accept: */*" -H "Content-Type: application/json" -d  
"{\"active\": true}"
```



### 4.4.1.3 Enable or Disable Device Model Synchronization with the Cloud

You can expose a local REST API to start or stop both the downstream and upstream device model synchronization. The API enables or disables the two flows consistently. No mixed scenarios are envisioned (for example, keep the upstream running and block only the downstream).

- API name: `deviceModelChannelActive`
- Body parameter: `active` (boolean)
- Example:

```
curl -X POST "http://<hostName>:<port>/<instanceId>/iot/edge/api/v1/admin/sync/deviceModelChannelActive" -H "accept: */*" -H "Content-Type: application/json" -d "{\"active\": true}"
```

### 4.4.1.4 Enable or Disable Data Model Synchronization with the Cloud

You can expose a local REST API to suspend and restart data model updates consumption from the Cloud.

- API name: `deviceModelChannelActive`
- Body parameter: `active` (boolean)
- Example:

```
curl -X POST "http://<hostName>:<port>/<instanceId>/iot/edge/api/v1/admin/sync/dataModelChannelActive" -H "accept: */*" -H "Content-Type: application/json" -d "{\"active\": true}"
```

## 4.4.2 Access using Swagger

If the Netty Server is enabled, the above APIs can be accessed through Swagger at the host and port defined in the file `sap.NettyConfiguration.cfg` (default localhost and port 8900) at the following address:

```
http://<host>:<port>/swagger-ui/index.html
```

## 4.4.3 Buffering on Internet of Things Edge Platform

The Internet of Things Edge Platform supports the buffering feature on the Edge component. This feature allows the Edge Platform to store messages that cannot be sent to SAP IoT due to missing connection.

Every time the Edge Platform loses connection with the cloud, either voluntarily (for instance, maintenance) or unintentionally (for instance, network failure), the messages will be stored in the file system of the machine on

which the Edge Platform is running. As soon as the connection is reestablished, all the stored messages will be sent to the cloud without losing any data.

To enable or disable, and to customize the buffering feature, edit the configuration file `config_gateway_{adapter}.xml`.

Below is an example for the MQTT adapter with the `config_gateway_mqtt.xml` file:

### Sample Code

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="cnfTransform.xsl"?>
<cnf:configuration xsi:schemaLocation="http://sap.com/iotservices/xsd/config"
  xmlns:cnf="http://sap.com/iotservices/xsd/config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...
  <cnf:gatewayBundle gatewayver="MQTT">
...
    <cnf:buffering active="true">
      <cnf:queueMaxElement>5000</cnf:queueMaxElement>
      <cnf:bufferTrigger>80</cnf:bufferTrigger>
      <cnf:maxFileDimension>1000</cnf:maxFileDimension>
      <cnf:maxDisk>4096000000</cnf:maxDisk>
      <cnf:saturation>70</cnf:saturation>
      <cnf:checkMemory>false</cnf:checkMemory>
    </cnf:buffering>
...
  </cnf:gatewayBundle>
...
</cnf:configuration>
```

The editable configurations of the buffering section are as listed below:

- **buffering active**: If set to **true**, it enables the buffering feature.
- **queueMaxElement**: Maximum number of messages in memory before storing them in the file system.
- **bufferTrigger**: Maximum percentage of usable memory before storing messages in the file system (available only with **checkMemory** parameter set to **true**).
- **maxFileDimension**: Maximum buffering file size (expressed in kilobyte). When the maximum size is reached, a new file will be created according to a rotation policy.
- **maxDisk**: Maximum file system space allocated to the buffering files (expressed in byte). When the maximum space is reached, a file will be deleted from the disk according to the FIFO policy.
- **saturation**: Maximum percentage of usable file system space before deleting the buffering files. When the maximum percentage is reached, a file will be deleted from the disk according to the FIFO policy.
- **checkMemory**: If set to **true**, it enables the usable memory check before storing messages in the file system (to use with the **bufferTrigger** parameter).

### Note

The default configuration provided with the Internet of Things Edge Platform is designed to be resilient and suitable for most of the scenarios. Changing the parameters without a thorough analysis could lead to unexpected data loss.

The buffering files are stored in the 'Buffering' directory, in the root folder of the Internet of Things Edge Platform, following the pattern: `Gateway/Buffering/...`

Below is a valid example:

Gateway/Buffering/DataPlaneManager/ipcMsgPrio\_Normal/2818295491935135

#### Note

The default storage folder is not customizable and must be considered only for internal use by the Internet of Things Edge Platform, in this case, no modification is allowed.

## 4.5 MQTT

The MQTT adapter is available for the cloud and for the edge. For more information how to use it with the Internet of Things Edge Platform, please refer to the tutorial [Set Up the Internet of Things Edge Platform - MQTT](#).

### Related Information

[Measure Message Format \(JSON\) \[page 59\]](#)

[Report the Measure Processing Results \(JSON\) \[page 70\]](#)

[Measure Payload Customization \[page 73\]](#)

[Command Message Format \(JSON\) \[page 74\]](#)

[Device Authorization and Authentication \[page 75\]](#)

[Endpoint Structure \[page 76\]](#)

### 4.5.1 Measure Message Format (JSON)

After a device has been successfully registered and a certain sensor has been assigned, the device can start sending messages that conform to a capability defined for the respective sensor type.

The schema of a valid measure message format looks as follows:

#### Sample Code

```
{
  "oneOf": [
    {
      "$ref": "#/definitions/messageObject"
    },
    {
      "type": "array",
      "items": {
        "$ref": "#/definitions/messageObject"
      }
    }
  ]
}
```

```

    ],
    "definitions":{
      "messageObject":{
        "title":"IoT Gateway JSON Message Schema",
        "type":"object",
        "properties":{
          "sensorAlternateId":{
            "id":"sensorAlternateId",
            "type":"string",
            "required":true,
            "description":"alternate id of the sensor the measures belong
to"
          },
          "capabilityAlternateId":{
            "id":"capabilityAlternateId",
            "type":"string",
            "required":true,
            "description":"alternate id of the capability the measures
conform to"
          },
          "timestamp":{
            "oneOf":[
              {
                "id":"timestamp",
                "type":"string",
                "required":false,
                "description":"time of the measure retrieval at the
device in ISO 8601 format"
              },
              {
                "id":"timestamp",
                "type":"number",
                "required":false,
                "description":"time of the measure retrieval at the
device in Unix format in milliseconds since January, 1st 1970 "
              }
            ]
          },
          "measureMessageId": {
            "id":"measureMessageId",
            "type":"string",
            "required":false,
            "description":"an identifier of the message with measurement
data, unique within the device"
          },
          "processingTag":{
            "id": "processingTag",
            "type": "string",
            "required": false,
            "description": "property for dispatching measures to the
corresponding Message Selector"
          },
          "measures":{
            "id":"measures",
            "type":"array",
            "minItems":1,
            "required":true,
            "items":{
              "oneOf":[
                {
                  "type":"array",
                  "minItems":1,
                  "required":true,
                  "items":{
                    "type":"string",
                    "description":"list of measures that conform to
the capability definition with the given alternate id"
                  }
                }
              ]
            }
          }
        }
      }
    }
  }
}

```



## Timestamp

For each measure, the timestamp can be reported. The Internet of Things Service use timestamp to order the measurement samples returned by the Internet of Things Service API. If a timestamp is not specified in the ingestion payload, the time of arrival of the message will be referenced.

The timestamp can be provided in two ways:

- **Message Timestamp**

As described in the schema above, you can specify the ingestion time for the whole message through the property `timestamp`.

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "timestamp": 1413191650000,
  "measures": [
    {
      "Humidity": 90,
      "Temperature": 23.5
    }
  ]
}
```

Both formats, ISO 8601 and a Unix time in milliseconds from epoch (that is a 13-digit number), are allowed.

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "timestamp": "2017-05-30T15:06:15.123Z",
  "measures": [
    {
      "Humidity": 90,
      "Temperature": 23.5
    }
  ]
}
```

- **Measure Timestamp**

- If you have modeled a capability with a property of type `date`, you can use it to specify the measure timestamp (that is the timestamp of the sample).

### Sample Code

```
{
  "sensorAlternateId": "sensor2",
  "capabilityAlternateId": "capability2",
  "measures": [
    {
      "Pressure": 90,
      "Temperature": 19.3,
      "Timestamp": 1413191650000
    }
  ]
}
```

Both formats, ISO 8601 and a Unix time in milliseconds from epoch (that is a 13-digit number), are allowed.

#### Sample Code

```
{
  "sensorAlternateId": "sensor2",
  "capabilityAlternateId": "capability2",
  "measures": [
    {
      "Pressure": 90,
      "Temperature": 19.3,
      "Timestamp": "2017-05-30T15:06:15.123Z"
    }
  ]
}
```

Looking at the example above, the payload includes a message that conforms to a capability with alternate ID `capability2`. The measures are for the sensor with the alternate ID `sensor2`. The measure contains three properties, namely `Pressure`, `Temperature`, and `Timestamp` (with the property `Timestamp` of type `date`). These fields are part of the respective capability definition. Modeling a property of type `date` is useful if:

- You want to use one of the properties of the device as timestamp of the measure.
- You want to send a single message with batched measures, for example, a message containing more samples for the capability, each one with a different timestamp (as for a default time series). For more information, please refer to section [Batched Measures \[page 19\]](#).
- You can use a special property named `'_time'` and not modelled into the capability to specify the measure timestamp (that is the timestamp of the sample).

#### Sample Code

```
{
  "sensorAlternateId": "sensor3",
  "capabilityAlternateId": "capability3",
  "measures": [
    {
      "Pressure": 90,
      "Temperature": 19.3,
      "_time": 1413191650000
    }
  ]
}
```

Both formats, ISO 8601 and a Unix time in milliseconds from epoch (that is a 13-digit number), are allowed.

#### Sample Code

```
{
  "sensorAlternateId": "sensor3",
  "capabilityAlternateId": "capability3",
  "measures": [
    {
      "Pressure": 90,
      "Temperature": 19.3,
      "_time": "2017-05-30T15:06:15.123Z"
    }
  ]
}
```

```
}
  ]
}
```

Looking at the example above, the payload includes a message that conforms to a capability with alternate ID `capability3`. The measures are for the sensor with the alternate ID `sensor3`. The measure contains three properties, namely `Pressure`, `Temperature`, and `_time`, but only `Pressure` and `Temperature` are part of the respective capability definition.

Using the special `_time` property is useful if:

- You want to use it as timestamp of the measure, without modelling it as a property of the capability.
- You want to send a single message with batched measures, for example, a message containing more samples for the capability, each one with a different timestamp (as for a default time series). For more information, please refer to section [Batched Measures \[page 19\]](#).

### Note

- Both, the Message Timestamp and the Measure Timestamp are normalized to Unix time in milliseconds from epoch (that is a 13-digit number) after the ingestion and represented as long.
- The Measure Timestamp has a higher priority than the Message Timestamp, for example, if both are present, the platform uses the Measure Timestamp as ingestion timestamp.
- The Measure Timestamp expressed via the special property "`_time`" has a higher priority to the Measure Timestamp expressed via a property of type 'date'; for example, if both are present, the platform uses the value of the `_time` property as ingestion timestamp.

## Processing Tag

A message can be tagged to be dispatched to the desired message selector. To do this, after have configured the processing of messages through the Message Processing API, you can specify the `processingTag` property in the message. Further information about the Internet of Things Message Processing, selectors, and their configuration can be found in section in the *Internet of Things Message Processing* documentation.

The following sample payload relies on a message selector using the `processingTag` property with value `sampleTag`.

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "processingTag": "sampleTag",
  "measures": [
    {
      "Status": "Good",
      "Humidity": 90
    }
  ]
}
```



## 4.5.1.2 Batched Measures

It is possible to send multiple measures that conform to the same capability in a single message.

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "measures": [
    {
      "Humidity": 90,
      "Temperature": 23.5,
      "Status": "Good"
    },
    {
      "Humidity": 93,
      "Temperature": 24.0,
      "Status": "Bad"
    }
  ]
}
```

## Timestamp

- **Message Timestamp**

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "timestamp": 1413191650000,
  "measures": [
    {
      "Humidity": 90,
      "Temperature": 23.5,
      "Status": "Good"
    },
    {
      "Humidity": 93,
      "Temperature": 24.0,
      "Status": "Bad"
    }
  ]
}
```

If you specify the timestamp as message timestamp, it will be applied to all the samples (for example, in this case both, the first and the second samples, will have the same ingestion timestamp 1413191650000)

- **Measure Timestamp**

- If you have modeled a property as type `date`, its value will result in the ingestion timestamp of each sample on the platform.

#### Sample Code

```
{
  "sensorAlternateId": "sensor2",
  "capabilityAlternateId": "capability2",
  "measures": [
    {
      "Pressure": 90,
      "Temperature": 19.3,
      "Timestamp": 1413191650000
    },
    {
      "Pressure": 75,
      "Temperature": 10.1,
      "Timestamp": 1413191790000
    }
  ]
}
```

For example, in this case you have two samples, the first one with 1413191650000 as ingestion timestamp and the second one with 1413191790000.

- If you have specified a `_time` property, without having modeled it as a property of the capability, its value will result in the ingestion timestamp of each sample on the platform.

#### Sample Code

```
{
  "sensorAlternateId": "sensor3",
  "capabilityAlternateId": "capability3",
  "measures": [
    {
      "Pressure": 90,
      "Temperature": 19.3,
      "_time": 1413191650000
    },
    {
      "Pressure": 75,
      "Temperature": 10.1,
      "_time": 1413191790000
    }
  ]
}
```

For example, in this case you have two samples, the first one with 1413191650000 as ingestion timestamp and the second one with 1413191790000.

### 4.5.1.3 Compressed Single Measures

It is possible to compress the measures by omitting the field names. In this case, the measures array is an array of arrays and the order must match the order of the properties in the capability definition.

#### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
```

```

    "measures": [
      [
        90,
        23.5,
        "Good"
      ]
    ]
  }

```

It is also possible to send null values for properties for which no values exist.

#### ↔ Sample Code

```

{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "measures": [
    [
      90,
      null,
      "Good"
    ]
  ]
}

```

### 4.5.1.4 Compressed Batched Measures

It is possible to send multiple measures with the compressed format (array) in a single message.

#### ↔ Sample Code

```

{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "measures": [
    [
      90,
      23.5,
      "Good"
    ],
    [
      94,
      24.0,
      "Bad"
    ]
  ]
}

```

### 4.5.1.5 Batched Measure Messages

It is possible to send a batch of messages that conform to different capabilities. In this case, the batch contains an array of messages.

### Sample Code

```
[
  {
    "sensorAlternateId": "sensor1",
    "capabilityAlternateId": "capability1",
    "measures": [
      {
        "Humidity": 90,
        "Temperature": 23.5,
        "Status": "Good"
      },
      {
        "Humidity": 93,
        "Temperature": 24.0,
        "Status": "Bad"
      }
    ]
  },
  {
    "sensorAlternateId": "sensor2",
    "capabilityAlternateId": "capability2",
    "measures": [
      {
        "Pressure": 90,
        "Temperature": 19.3,
        "Timestamp": 1413191650000
      }
    ]
  }
]
```

## 4.5.1.6 Measures for the Auto-Onboarding of Sensors

It is possible to auto-onboard sensors during the data ingestion. Auto-onboarding means that sensors that do not exist are created automatically during data ingestion. In this case, an additional attribute, namely the `sensorTypeAlternateId`, must be passed. If the sensor with the given `sensorAlternateId` does not exist, a new sensor is created for the device that is assigned to the sensor type with this `sensorTypeAlternateId`.

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "sensorTypeAlternateId": "12",
  "capabilityAlternateId": "capability1",
  "measures": [
    {
      "Humidity": 90,
      "Temperature": 23.5,
      "Status": "Good"
    }
  ]
}
```

### Note

Only numeric values are supported for the alternate ID of a sensor type.

### Note

The following error codes are returned for failing auto-onboarding attempts against the Internet of Things Gateway Cloud REST:

- 401 if the device does not exist.
- 400 if the message format is not correct, for example, one of the `sensorAlternateId`, `sensorTypeAlternateId`, or `capabilityAlternateId` is missing.

## 4.5.1.7 Collecting Measurements for Multiple Devices

You can send measurements related to multiple devices in a single payload.

### Prerequisites

The endpoint in which you send the payload must be related to a router device.

An additional property `deviceAlternateId` is specified in the payload.

### How to enable

Create, if it does not exist already, a new file `sap.IngestionConfiguration.cfg` under the `config/services` directory.

Add a new line with the following content:

```
allowMultipleDevicesInPayload=true
```

### Example

To run this example, it is necessary to have:

- A capability with the alternate ID `temperature_capability`.
- The capability contains one property: Temperature of type `float`.
- A sensor type with alternate ID 345 with the previously mentioned capability assigned with type `measure`.
- A router device.

The following payload sent into the endpoint `measures/<routerDeviceAlternateId>` will produce the below result:

- Device 0 will send the following temperature value: 25
- Device 1 will send the following temperature value: 13.2

## Sample Code

```
[
  {
    "deviceAlternateId": "<deviceAlternateId_0>",
    "capabilityAlternateId": "temperature_capability",
    "sensorTypeAlternateId": "345",
    "measures": [
      {
        "Temperature": 25
      }
    ]
  },
  {
    "deviceAlternateId": "<deviceAlternateId_1>",
    "capabilityAlternateId": "temperature_capability",
    "sensorTypeAlternateId": "345",
    "measures": [
      {
        "Temperature": 13.2
      }
    ]
  }
]
```

All the sample payloads shown in the Measure Message Format (JSON) section are valid; you just need to add the `deviceAlternateId` that collected the measure.

## 4.5.2 Report the Measure Processing Results (JSON)

The Internet of Things Gateway reports the processing results for the posted measures to the device by publishing corresponding records to the topic `ack/<device alternate ID>`.

Note that the Internet of Things Gateway reports the processing status immediately, before forwarding the measures to the Internet of Things Service. If the Internet of Things Gateway considers the measure message invalid and rejects it, this outcome is final. However, if the Internet of Things Gateway accepts the measure message as valid, it may still be rejected or lost further down the road. Thus, the Internet of Things Gateway typically does not return the `200 OK` status code, but only `202 Accepted`. The MQTT API will be extended in future releases of the gateway, allowing HTTP clients to query the final processing outcome of measure messages posted earlier, including the processing by the Internet of Things Service.

### 4.5.2.1 Response Message

The detailed status message is a list of objects. The number of objects is either one or corresponds to the number of measure messages posted within [Batched Measure Messages \[page 22\]](#). In the latter case, the order of the objects is the same as the order of the measure messages in the batch. Each object can have the following fields:

```
{
  "id": "optional; the value of the measureMessageId field if the device has
  provided it when posting the measurement data",
```

```

"code": "the status code for this Measure Message; the same as the HTTP status
code if there is only one Measure Message in the HTTP request",
"sensorAlternateId": "value from the request",
"capabilityAlternateId": "value from the request",
"messages": [a list of human-readable strings, mostly error messages, generated
during the verification of the measure message]
}

```

Subsequent versions of the Internet of Things Gateway may extend the object with additional fields. All fields except for the `code` field can be missing (for example, if the JSON request could not be parsed, or no errors were detected).

If the device needs to bind the objects in the response message to the corresponding measure messages explicitly, it should provide unique values of `measureMessageId` field for every measure message. These values are returned in the `id` field of the matching response messages.

## 4.5.2.2 Processing Status Codes

The semantics of the `code` field is the same as of the corresponding HTTP status code (see the following table).

Status code	Meaning
200 OK	The Internet of Things Gateway has received a confirmation from the Internet of Things Service that the measure message was processed correctly before returning the result to the device. Almost never occurs.
202 Accepted	The measure message has passed the validation check executed by the Internet of Things Gateway.
400 Bad Request	The measure message has failed the validation check executed by the Internet of Things Gateway. If the message posted is a batched message, all message parts have failed the validation check. All measures were discarded. The body of the response contains detailed information.
401 Unauthorized	Authentication has failed, or the authenticated device is not authorized to post measures on behalf of the device with the alternate ID specified in the URL. For more information, please refer to section <a href="#">Device Authorization and Authentication [page 26]</a> .
500 Internal Server Error	The Internet of Things Gateway is not configured correctly, or a software error has occurred in the Internet of Things Gateway.

## 4.5.2.3 Error Messages

The `messages` field can contain any human-readable error messages. There is no fixed list of allowed values, but the messages may start with the strings listed in the following table.

Message	Condition
<code>Unauthorized</code>	The client is not authorized to post the measure messages on behalf of the device. In particular, this may mean an HTTP authentication failure.
<code>Invalid device alternate ID</code>	Most likely to appear when a valid and authenticated router device posts measure messages on behalf of another one.
<code>Invalid message format</code>	This means that the whole payload could not be parsed, for example due to invalid JSON syntax. The sensor and device alternate IDs, etc., cannot be extracted in this case.
<code>Invalid sensor alternate ID</code>	The sensor does not exist and could not be onboarded automatically.
<code>Invalid capability alternate ID</code>	The capability does not exist for the sensor, or it does not accept measures (being of the command type).
<code>Invalid property alternate ID</code>	The property does not exist in the capability.
<code>Illegal property value</code>	The value is not acceptable for the reason specified in the same error message. The message can be used if the measure is posted as a list with a wrong number of elements. It can also be used when the value for a certain property cannot be converted to the corresponding type.
<code>No valid measurement data found</code>	No property of the capability was assigned a valid value.
<code>Illegal time stamp value</code>	The time stamp provided by the client for the measure message could not be parsed.
<code>Gateway configuration error</code>	Typically returned for a wrongly configured Internet of Things Gateway Edge. However, it may also be sent if there are errors in the data model, such as a capability without any properties.
<code>Gateway internal error</code>	Some other error was detected, and an internal consistency check has failed in the Internet of Things Gateway.



## 4.5.3 Measure Payload Customization

It is possible to use custom measure message format to inject measures into the Internet of Things Edge Platform MQTT.

### Prerequisites

The custom payload must be in JSON format.

### How to Enable

An apache freemarker template must be provided to transform the custom payload into a payload that respects the standard measure format. This will enable the Internet of Things Edge Platform to ingest custom payloads.

The template must be named `measureTemplate.ftl` and placed under the `config/freemarker` directory in the gateway folder.

### Example

To run this example, it is necessary to have:

A capability with the alternate ID `temperature_capability`.

The capability contains one property: Temperature of type float.

A sensor type with alternate ID 345 with the previously mentioned capability assigned with type measure.

Starting from the following custom payload:

#### Sample Code

```
{
  "time": 1604509375000,
  "sensor": "my-sensor",
  "value": 11.3
}
```

You can transform it into a standard measure by applying the following sample template:

#### Sample Code

```
<#escape output as output?json_string> (1)
{
  "sensorAlternateId": "${data.sensor}",
  "sensorTypeAlternateId": "345", (3)
```

```

"capabilityAlternateId": "temperature_capability", (3)
<#if data.time??> (2)
"timestamp": #{data.time},
</#if>
"measures": [
  {
    "Temperature": #{data.value}
  }
]
}
</#escape>

```

#### 📌 Note

1. Ensure json escape is applied to the output.
2. Add the information only if time attribute is present in the original payload.
3. Fixed value for `sensorTypeAlternateId` and `capabilityAlternateId`.

## External Tools

If you need to try your template, you can use the following online tester from Apache foundation:

<https://try.freemarker.apache.org/> ➔

#### 📌 Note

In the DataModel area, put `data=<your json>`

You must select `angleBracket` in the Tag Syntax input.

You must select `legacy` in the Interpolation Syntax input.

## 4.5.4 Command Message Format (JSON)

After a device has been successfully registered and a certain sensor has been assigned, the device can start receiving commands that conform to a capability defined for the respective sensor type.

The schema of a valid command JSON payload looks as follows:

```

{
  "title": "IoT Gateway JSON Message Schema",
  "type": "object",
  "properties": {
    "sensorAlternateId": {
      "id": "sensorAlternateId",
      "type": "string",
      "required": true,
      "description": "alternate id of the sensor the measures belong to"
    },
    "capabilityAlternateId": {
      "id": "capabilityAlternateId",
      "type": "string",

```

```

    "required":true,
    "description":"alternate id of the capability the command conform to"
  },
  "command":{
    "id":"command",
    "type":"object",
    "description":"command that conform to the capability definition with
the given alternate id"
  }
}
}

```

The following is an example of a valid command message published by the Internet of Things Gateway. A device can consume the message, by subscribing to the commands topic as described in the section *Endpoint Structure*.

#### Sample Code

```

{
  "sensorAlternateId":"sensor2",
  "capabilityAlternateId":"capability2",
  "command":{
    "LED":"on",
    "Buzzer":"off",
    "Speed":45
  }
}

```

The payload includes a message that conforms to a capability with the alternate ID `capability2`. The command is for the sensor with the alternate ID `sensor2`. The command contains three properties, namely `LED`, `Buzzer`, and `Speed`.

## 4.5.5 Device Authorization and Authentication

The client authorization logic cross-checks the metadata provided by the connecting device through its client certificate with the resources the connecting device is trying to access.

In each client connection, the following authorization checks are performed:

- The `instanceId` included in the certificate must match the ID of the instance to which the Internet of Things Gateway belongs.
- The `deviceAlternateId` included in the certificate must match the ID of the instance to which the Internet of Things Gateway belongs.
- The `deviceAlternateId` included in the certificate must match the MQTT `clientId` retrieved from the MQTT metadata. This constraint can be relaxed in case the connecting device has the authorization type `router`, as described in section [Connection as a Router Device \[page 27\]](#).

If an MQTT client attempts to post measurement data for ingestion, the following authorization checks are performed:

- The topic to which the measurement data is published must match the structure that is defined for the data ingestion topic. For more information, please refer to section [Measure Message Format \(JSON\) \[page 13\]](#).
- The `deviceAlternateId` included in the certificate must match the `alternateId` value that is specified in the measure ingestion topic.

If an MQTT client attempts to subscribe to the command topic, the following authorization checks are performed:

- The target topic must match the structure that is defined for the command topic. For more information, please refer to section [Command Message Format \(JSON\) \[page 25\]](#).
- The `deviceAlternateId` included in the certificate must match the `alternateId` value that is specified in the command topic.

## 4.5.6 Endpoint Structure

### 4.5.6.1 Secure Transport

The port for the Internet of Things Edge Platform MQTT with secure transport is 61628. Therefore, the server URI for the Internet of Things Edge Platform for MQTT is:

```
ssl://<IOT_GATEWAY_IP>:61628
```

The specific configuration of the MQTT service for data ingestion exposed by Internet of Things Edge Platform MQTT relies on the configuration file `<GatewayEdge_deploy_folder>/config/config_gateway_mqtt.xml`.

The file has the following standard format:

```
<cnf:transportConnectors>
  <cnf:transportConnector name="mqtt" uri="mqtt://127.0.0.1:61618?
transport.soTimeout=60000"/>
  <cnf:transportConnector name="mqtt+mutualAuth" uri="mqtt+nio+ssl://
127.0.0.1:61628?wantClientAuth=true&transport.soTimeout=60000"/>
</cnf:transportConnectors>
```

The following steps are required to run Internet of Things Edge Platform MQTT with the secure transport enabled.

1. Set the URI parameter in the `<cnf:transportConnector>` tag, with `mqtt+mutualAuth` name, to the IP of the system where the Internet of Things Edge Platform is running.

```
<cnf:transportConnectors>
  ...
  <cnf:transportConnector name="mqtt+mutualAuth" uri="mqtt+nio+ssl://
127.0.0.1:61628?wantClientAuth=true&transport.soTimeout=60000"/>
  ...
</cnf:transportConnectors>
```

#### Note

The default secure port for the MQTT API is 61628 (if necessary, change it by setting the value in the URI).

2. Start the Internet of Things Edge Platform MQTT by launching `gateway.bat / .sh`

## 4.5.6.1.1 Custom Certificates

### Overview

It is possible to change the MQTT Edge Platform default behavior for publishing the MQTT SSL transports using a custom certificate, instead of relying on the SAP IoT certificates.

In this way, the customers can use their own certificates to expose the MQTT secure transports using, for example, a certificate signed from a trusted certificate authority.

### How to Enable

To enable this feature, the MQTT Edge Platform must be configured in the following way:

- Prepare a `.p12` file with the custom certificate to be used for publishing the MQTT transports.
- Store it into the Internet of Things Edge Platform file system, under the folder `'config/certificates/server'` (if the folder does not exist, create it).
- Go to the `'sap.BrokerConfiguration.cfg'` config file (under the folder `'config/services'`) and:
  - Enable the usage of the custom certificate, setting the property `useCustomServerCertificate` to `true`.
  - Configure the path to the custom certificate, via the property `customKeyStoreFilePath`.
  - Configure the secret required to open the custom certificate, in one of these two ways:
    - Setting the secret via the property `customKeyStoreSecret`.
    - Passing the secret to the Internet of Things Edge Platform via a System Variable, and setting the name of that variable via the property `customKeyStoreSecretEnvVariable`.

Given below is a `'sap.BrokerConfiguration.cfg'` file template, specifying the secret directly into the file:

#### Sample Code

```
# Enable usage of a custom certificate
useCustomServerCertificate = true
# Sub path where the custom certificate is stored
customKeyStoreFilePath = config/certificates/server/my-certificate.p12
# Secret for custom certificate
customKeyStoreSecret = p6N9NUm87MxGUvm9
```

Here is a `'sap.BrokerConfiguration.cfg'` file template, passing the secret via System Variable:

#### Sample Code

```
# Enable usage of a custom certificate
```

```
useCustomServerCertificate = true
# Sub path where the custom certificate is stored
customKeyStoreFilePath = config/certificates/server/my-certificate.p12
# Name of the system variable containing the secret for custom certificate
customKeyStoreSecretEnvVariable = myEnvVariableForSecret
```

### 4.5.6.1.1.1 Prerequisites

Please follow the below prerequisites to ensure that the feature works properly and avoid any unexpected behavior.

#### Certificate's Format and Secret

- The certificate must be stored as .p12 file (other formats are not supported).
- The secret to open the .p12 file cannot be empty.

#### Certificate's Health and Security Checks

The Internet of Things Edge Platform reserves the right to validate the certificate before using it; in case it does not satisfy the security checks, the publication of the SSL transports will fail.

Here are the main requirements for trusted certificates:

- TLS server certificates and issuing CAs using RSA keys must use key sizes greater than or equal to 2048 bits. Certificates using RSA key sizes smaller than 2048 bits are not valid.
- TLS server certificates and issuing CAs must use a hash algorithm from the SHA-2 family in the signature algorithm. SHA-1 signed certificates are not valid.
- TLS server certificates must not be expired.

#### Certificate Renewal

##### ⓘ Note

The customer is responsible for the certificate renewal, and must take care of configuring a new custom certificate on the Internet of Things Edge Platform, before the previous one is expired. A restart of the MQTT transports will be required to make the renewal effective.

Here are the main steps for renewing a certificate:

- Store the new certificate in the Internet of Things Edge Platform file system, under the folder 'config/certificates/server'.

- Change the 'sap.BrokerConfiguration.cfg' file (under the folder 'config/services'), updating the field 'customKeyStoreFilePath' to point to the new file (as a result of this, the Internet of Things Edge Platform will automatically restart the MQTT transports).
- Update the certificate secret, if changed (setting the value of the new secret via the config file or updating the value of the system variable, as described before).

## 4.5.6.1.1.2 Device Authorization

Once the MQTT Edge Platform is configured to publish the secure transports, by default it publishes them requiring the mutual auth (that is, the client certificate is mandatory) and using the client certificate metadata to check if the device is authorized to connect or not. As result, a device can be authorized only if it uses a certificate released by SAP IoT Service. For this reason, to keep the device authorization enabled, the devices must continue using SAP IoT certificates, even if the Internet of Things Edge Platform is configured for publishing the MQTT secure transports with a custom certificate.

If the customer is willing to disable the device authorization and rely only on the device authentication given by the usage of a trusted certificate, the customer can configure the Internet of Things Edge Platform to accept connections from devices using a custom certificate.

The main steps to do this are:

- Go to the 'gateway.sh/.bat' config file (under the main Internet of Things Edge Platform folder) and disable the device authorization checks, setting the VMArg `com.sap.iotservices.useDeviceAuth` to `false` as shown below:  
`-Dcom.sap.iotservices.useDeviceAuth=false`
- If the device certificate is signed from a custom CA, that CA must be added to the Internet of Things Edge Platform truststore as per the below steps:
  - Prepare a `.p12` file containing the required CAs.
  - Store it into the Internet of Things Edge Platform file system, under the folder 'config/certificates/server'.
  - Go to the 'sap.BrokerConfiguration.cfg' config file (under the folder 'config/services') and:
    - Configure the path to the custom CA, setting the property `customTrustStoreFilePath`.
    - Configure the secret required to open the custom CA, in one of these two ways:
      - Setting the secret via the property `customTrustStoreSecret`.
      - Passing the secret to the Internet of Things Edge Platform via a System Variable, and setting the name of that variable via the property `customTrustStoreSecretEnvVariable`.

Given below is a 'sap.BrokerConfiguration.cfg' file template, specifying the CA secret directly into the file:

### Sample Code

```
# Enable usage of a custom certificate
useCustomServerCertificate = true
# Sub path where the custom certificate is stored
customKeyStoreFilePath = config/certificates/server/my-certificate.p12
# Secret for custom certificate
customKeyStoreSecret = p6N9NUm87MxGUvm9
# Sub path where the custom CA is stored
customTrustStoreFilePath = config/certificates/server/my-ca.p12
```

```
# Secret for custom CA
customTrustStoreSecret = caSecret!
```

Given below is a 'sap.BrokerConfiguration.cfg' file template, passing the secret via System Variable:

### Sample Code

```
# Enable usage of a custom certificate
useCustomServerCertificate = true
# Sub path where the custom certificate is stored
customKeyStoreFilePath = config/certificates/server/my-certificate.p12
# Name of the system variable containing the secret for custom certificate
customKeyStoreSecretEnvVariable = myEnvVariableForSecret
# Sub path where the custom CA is stored
customTrustStoreFilePath = config/certificates/server/my-ca.p12
# Name of the system variable containing the secret for custom CA
customTrustStoreSecretEnvVariable = customTrustStoreSecret
```

### Note

In case a custom CA is configured:

- Its public key must be stored into a .p12 file (other formats are not supported).
- The secret required to open the .p12 file cannot be empty.
- The public key contained in the .p12 file must satisfy the same security requirements as mentioned in the section "Certificate's Health and Security Checks" in [Prerequisites \[page 78\]](#).
- The customer is responsible for the CA renewal and must take care of configuring a new CA on the Internet of Things Edge Platform, before the previous one is expired. A restart of the MQTT transports will be required to make the renewal effective.

The main steps for renewing a CA are listed below:

- Store the new CA file in the Internet of Things Edge Platform file system, under the folder 'config/certificates/server'.
- Change the 'sap.BrokerConfiguration.cfg' file (under the folder 'config/services'), updating the field 'customTrustStoreFilePath' to point to the new file (as a result of this, the Internet of Things Edge Platform will automatically restart the MQTT endPoints).
- Update the certificate secret, if changed (setting the value of the new secret via the config file or updating the value of the system variable, as described before).

## 4.5.6.2 Insecure Transport

The port for the Internet of Things Edge Platform MQTT with insecure transport is 61618. Therefore, the server URI for the Internet of Things Gateway Edge for MQTT is:

```
tcp://<IOT_GATEWAY_IP>:61618
```

Measures can be ingested into the Internet of Things Edge Platform MQTT by publishing data on a topic with the following structure:

```
measures/<deviceAlternateId>
```



The `deviceAlternateId` represents the alternate ID of the device that generated the measures.

Commands can be delivered to devices on the following topic:

```
commands/<deviceAlternateId>
```

The `deviceAlternateId` represents the alternate ID of the device to which the commands are directed.

## 4.6 REST

The REST adapter is available for the cloud and for the edge. For more information how to use it with the Internet of Things Edge Platform, please refer to the tutorial [Set Up the Internet of Things Edge Platform - REST](#).

### Related Information

- [Measure Message Format \(JSON\) \[page 81\]](#)
- [Report the Measure Processing Results \(JSON\) \[page 92\]](#)
- [Measure Payload Customization \[page 95\]](#)
- [Command Message Format \(JSON\) \[page 97\]](#)
- [Measure Message Format \(Protobuf\) \[page 98\]](#)
- [Report the Measure Processing Results \(Protobuf\) \[page 101\]](#)
- [Command Message Format \(Protobuf\) \[page 103\]](#)
- [Device Authorization and Authentication \[page 104\]](#)
- [Endpoint Structure \[page 105\]](#)

### 4.6.1 Measure Message Format (JSON)

After a device has been successfully registered and a certain sensor has been assigned, the device can start sending messages that conform to a capability defined for the respective sensor type.

The schema of a valid measure message format looks as follows:

#### Sample Code

```
{
  "oneOf": [
    {
      "$ref": "#/definitions/messageObject"
    },
    {
      "type": "array",
      "items": {
```

```

    "$ref": "#/definitions/messageObject"
  }
},
"definitions": {
  "messageObject": {
    "title": "IoT Gateway JSON Message Schema",
    "type": "object",
    "properties": {
      "sensorAlternateId": {
        "id": "sensorAlternateId",
        "type": "string",
        "required": true,
        "description": "alternate id of the sensor the measures belong
to"
      },
      "capabilityAlternateId": {
        "id": "capabilityAlternateId",
        "type": "string",
        "required": true,
        "description": "alternate id of the capability the measures
conform to"
      },
      "timestamp": {
        "oneOf": [
          {
            "id": "timestamp",
            "type": "string",
            "required": false,
            "description": "time of the measure retrieval at the
device in ISO 8601 format"
          },
          {
            "id": "timestamp",
            "type": "number",
            "required": false,
            "description": "time of the measure retrieval at the
device in Unix format in milliseconds since January, 1st 1970 "
          }
        ]
      },
      "measureMessageId": {
        "id": "measureMessageId",
        "type": "string",
        "required": false,
        "description": "an identifier of the message with measurement
data, unique within the device"
      },
      "processingTag": {
        "id": "processingTag",
        "type": "string",
        "required": false,
        "description": "property for dispatching measures to the
corresponding Message Selector"
      },
      "measures": {
        "id": "measures",
        "type": "array",
        "minItems": 1,
        "required": true,
        "items": {
          "oneOf": [
            {
              "type": "array",
              "minItems": 1,
              "required": true,
              "items": {
                "type": "string",

```



## Timestamp

For each measure, the timestamp can be reported. The Internet of Things Service use timestamp to order the measurement samples returned by the Internet of Things Service API. If a timestamp is not specified in the ingestion payload, the time of arrival of the message will be referenced.

The timestamp can be provided in two ways:

- **Message Timestamp**

As described in the schema above, you can specify the ingestion time for the whole message through the property `timestamp`.

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "timestamp": 1413191650000,
  "measures": [
    {
      "Humidity": 90,
      "Temperature": 23.5
    }
  ]
}
```

Both formats, ISO 8601 and a Unix time in milliseconds from epoch (that is a 13-digit number), are allowed.

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "timestamp": "2017-05-30T15:06:15.123Z",
  "measures": [
    {
      "Humidity": 90,
      "Temperature": 23.5
    }
  ]
}
```

- **Measure Timestamp**

- If you have modeled a capability with a property of type `date`, you can use it to specify the measure timestamp (that is the timestamp of the sample).

### Sample Code

```
{
  "sensorAlternateId": "sensor2",
  "capabilityAlternateId": "capability2",
  "measures": [
    {
      "Pressure": 90,
      "Temperature": 19.3,
      "Timestamp": 1413191650000
    }
  ]
}
```

Both formats, ISO 8601 and a Unix time in milliseconds from epoch (that is a 13-digit number), are allowed.

#### Sample Code

```
{
  "sensorAlternateId": "sensor2",
  "capabilityAlternateId": "capability2",
  "measures": [
    {
      "Pressure": 90,
      "Temperature": 19.3,
      "Timestamp": "2017-05-30T15:06:15.123Z"
    }
  ]
}
```

Looking at the example above, the payload includes a message that conforms to a capability with alternate ID `capability2`. The measures are for the sensor with the alternate ID `sensor2`. The measure contains three properties, namely `Pressure`, `Temperature`, and `Timestamp` (with the property `Timestamp` of type `date`). These fields are part of the respective capability definition. Modeling a property of type `date` is useful if:

- You want to use one of the properties of the device as timestamp of the measure.
- You want to send a single message with batched measures, for example, a message containing more samples for the capability, each one with a different timestamp (as for a default time series). For more information, please refer to section [Batched Measures \[page 19\]](#).
- You can use a special property named `'_time'` and not modelled into the capability to specify the measure timestamp (that is the timestamp of the sample).

#### Sample Code

```
{
  "sensorAlternateId": "sensor3",
  "capabilityAlternateId": "capability3",
  "measures": [
    {
      "Pressure": 90,
      "Temperature": 19.3,
      "_time": 1413191650000
    }
  ]
}
```

Both formats, ISO 8601 and a Unix time in milliseconds from epoch (that is a 13-digit number), are allowed.

#### Sample Code

```
{
  "sensorAlternateId": "sensor3",
  "capabilityAlternateId": "capability3",
  "measures": [
    {
      "Pressure": 90,
      "Temperature": 19.3,
      "_time": "2017-05-30T15:06:15.123Z"
    }
  ]
}
```

```
}
  ]
}
```

Looking at the example above, the payload includes a message that conforms to a capability with alternate ID `capability3`. The measures are for the sensor with the alternate ID `sensor3`. The measure contains three properties, namely `Pressure`, `Temperature`, and `_time`, but only `Pressure` and `Temperature` are part of the respective capability definition.

Using the special `_time` property is useful if:

- You want to use it as timestamp of the measure, without modelling it as a property of the capability.
- You want to send a single message with batched measures, for example, a message containing more samples for the capability, each one with a different timestamp (as for a default time series). For more information, please refer to section [Batched Measures \[page 19\]](#).

### Note

- Both, the Message Timestamp and the Measure Timestamp are normalized to Unix time in milliseconds from epoch (that is a 13-digit number) after the ingestion and represented as long.
- The Measure Timestamp has a higher priority than the Message Timestamp, for example, if both are present, the platform uses the Measure Timestamp as ingestion timestamp.
- The Measure Timestamp expressed via the special property `"_time"` has a higher priority to the Measure Timestamp expressed via a property of type `'date'`; for example, if both are present, the platform uses the value of the `_time` property as ingestion timestamp.

## Processing Tag

A message can be tagged to be dispatched to the desired message selector. To do this, after have configured the processing of messages through the Message Processing API, you can specify the `processingTag` property in the message. Further information about the Internet of Things Message Processing, selectors, and their configuration can be found in section in the *Internet of Things Message Processing* documentation.

The following sample payload relies on a message selector using the `processingTag` property with value `sampleTag`.

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "processingTag": "sampleTag",
  "measures": [
    {
      "Status": "Good",
      "Humidity": 90
    }
  ]
}
```

## 4.6.1.2 Batched Measures

It is possible to send multiple measures that conform to the same capability in a single message.

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "measures": [
    {
      "Humidity": 90,
      "Temperature": 23.5,
      "Status": "Good"
    },
    {
      "Humidity": 93,
      "Temperature": 24.0,
      "Status": "Bad"
    }
  ]
}
```

## Timestamp

- **Message Timestamp**

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "timestamp": 1413191650000,
  "measures": [
    {
      "Humidity": 90,
      "Temperature": 23.5,
      "Status": "Good"
    },
    {
      "Humidity": 93,
      "Temperature": 24.0,
      "Status": "Bad"
    }
  ]
}
```

If you specify the timestamp as message timestamp, it will be applied to all the samples (for example, in this case both, the first and the second samples, will have the same ingestion timestamp 1413191650000)

- **Measure Timestamp**

- If you have modeled a property as type `date`, its value will result in the ingestion timestamp of each sample on the platform.

#### Sample Code

```
{
  "sensorAlternateId": "sensor2",
  "capabilityAlternateId": "capability2",
  "measures": [
    {
      "Pressure": 90,
      "Temperature": 19.3,
      "Timestamp": 1413191650000
    },
    {
      "Pressure": 75,
      "Temperature": 10.1,
      "Timestamp": 1413191790000
    }
  ]
}
```

For example, in this case you have two samples, the first one with 1413191650000 as ingestion timestamp and the second one with 1413191790000.

- If you have specified a `_time` property, without having modeled it as a property of the capability, its value will result in the ingestion timestamp of each sample on the platform.

#### Sample Code

```
{
  "sensorAlternateId": "sensor3",
  "capabilityAlternateId": "capability3",
  "measures": [
    {
      "Pressure": 90,
      "Temperature": 19.3,
      "_time": 1413191650000
    },
    {
      "Pressure": 75,
      "Temperature": 10.1,
      "_time": 1413191790000
    }
  ]
}
```

For example, in this case you have two samples, the first one with 1413191650000 as ingestion timestamp and the second one with 1413191790000.

### 4.6.1.3 Compressed Single Measures

It is possible to compress the measures by omitting the field names. In this case, the measures array is an array of arrays and the order must match the order of the properties in the capability definition.

#### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
```



```
"measures": [
  [
    90,
    23.5,
    "Good"
  ]
]
```

It is also possible to send null values for properties for which no values exist.

#### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "measures": [
    [
      90,
      null,
      "Good"
    ]
  ]
}
```

### 4.6.1.4 Compressed Batched Measures

It is possible to send multiple measures with the compressed format (array) in a single message.

#### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "capabilityAlternateId": "capability1",
  "measures": [
    [
      90,
      23.5,
      "Good"
    ],
    [
      94,
      24.0,
      "Bad"
    ]
  ]
}
```

### 4.6.1.5 Batched Measure Messages

It is possible to send a batch of messages that conform to different capabilities. In this case, the batch contains an array of messages.

### Sample Code

```
[
  {
    "sensorAlternateId": "sensor1",
    "capabilityAlternateId": "capability1",
    "measures": [
      {
        "Humidity": 90,
        "Temperature": 23.5,
        "Status": "Good"
      },
      {
        "Humidity": 93,
        "Temperature": 24.0,
        "Status": "Bad"
      }
    ]
  },
  {
    "sensorAlternateId": "sensor2",
    "capabilityAlternateId": "capability2",
    "measures": [
      {
        "Pressure": 90,
        "Temperature": 19.3,
        "Timestamp": 1413191650000
      }
    ]
  }
]
```

## 4.6.1.6 Measures for the Auto-Onboarding of Sensors

It is possible to auto-onboard sensors during the data ingestion. Auto-onboarding means that sensors that do not exist are created automatically during data ingestion. In this case, an additional attribute, namely the `sensorTypeAlternateId`, must be passed. If the sensor with the given `sensorAlternateId` does not exist, a new sensor is created for the device that is assigned to the sensor type with this `sensorTypeAlternateId`.

### Sample Code

```
{
  "sensorAlternateId": "sensor1",
  "sensorTypeAlternateId": "12",
  "capabilityAlternateId": "capability1",
  "measures": [
    {
      "Humidity": 90,
      "Temperature": 23.5,
      "Status": "Good"
    }
  ]
}
```

### Note

Only numeric values are supported for the alternate ID of a sensor type.

### Note

The following error codes are returned for failing auto-onboarding attempts against the Internet of Things Gateway Cloud REST:

- 401 if the device does not exist.
- 400 if the message format is not correct, for example, one of the `sensorAlternateId`, `sensorTypeAlternateId`, or `capabilityAlternateId` is missing.

## 4.6.1.7 Collecting Measurements for Multiple Devices

You can send measurements related to multiple devices in a single payload.

### Prerequisites

The endpoint in which you send the payload must be related to a router device.

An additional property `deviceAlternateId` is specified in the payload.

### How to enable

Create, if it does not exist already, a new file `sap.IngestionConfiguration.cfg` under the `config/services` directory.

Add a new line with the following content:

```
allowMultipleDevicesInPayload=true
```

### Example

To run this example, it is necessary to have:

- A capability with the alternate ID `temperature_capability`.
- The capability contains one property: Temperature of type `float`.
- A sensor type with alternate ID 345 with the previously mentioned capability assigned with type `measure`.
- A router device.

The following payload sent into the endpoint `measures/<routerDeviceAlternateId>` will produce the below result:

- Device 0 will send the following temperature value: 25
- Device 1 will send the following temperature value: 13.2

## Sample Code

```
[
  {
    "deviceAlternateId": "<deviceAlternateId_0>",
    "capabilityAlternateId": "temperature_capability",
    "sensorTypeAlternateId": "345",
    "measures": [
      {
        "Temperature": 25
      }
    ]
  },
  {
    "deviceAlternateId": "<deviceAlternateId_1>",
    "capabilityAlternateId": "temperature_capability",
    "sensorTypeAlternateId": "345",
    "measures": [
      {
        "Temperature": 13.2
      }
    ]
  }
]
```

All the sample payloads shown in the Measure Message Format (JSON) section are valid; you just need to add the `deviceAlternateId` that collected the measure.

## 4.6.2 Report the Measure Processing Results (JSON)

The Internet of Things Gateway reports the processing results for the posted measures to the device using the REST API. The general outcome is reported as the status code returned for the corresponding HTTP request. The details including the error messages are returned in the body of the HTTP response which has the same content type as the request: if the measures are posted in the JSON format, the detailed processing results are also returned in JSON. If the measures are posted in the Google Protocol Buffers format, the results are returned in protobuf. For more information, please refer to section [Report the Measure Processing Results \(Protobuf\) \[page 44\]](#).

Note that the Internet of Things Gateway reports the processing status immediately, before forwarding the measures to the Internet of Things Service. If the Internet of Things Gateway considers the measure message invalid and rejects it, this outcome is final. However, if the Internet of Things Gateway accepts the measure message as valid, it may still be rejected or lost further down the road. Thus, the Internet of Things Gateway typically does not return the 200 OK status code, but only 202 Accepted. The REST API will be extended in future releases of the gateway, allowing HTTP clients to query the final processing outcome of measure messages posted earlier, including the processing by the Internet of Things Service.

### 4.6.2.1 HTTP Status Codes

The device must be able to handle at least the status codes described in the following table. This list is not exhaustive. Future versions of the Internet of Things Gateway may return additional status codes.

Status code	Meaning
200 OK	The Internet of Things Gateway has received a confirmation from the Internet of Things Service that the measure message was processed correctly before returning the result to the device. Almost never occurs.
202 Accepted	The measure message has passed the validation check executed by the Internet of Things Gateway.
207 Multi-Status	<a href="#">Batched Measure Messages [page 22]</a> were posted to the Internet of Things Gateway. Some messages in the batch have failed the validation, but at least one was accepted. The body of the response contains detailed information.
400 Bad Request	The measure message has failed the validation check executed by the Internet of Things Gateway. If the message posted is a batched message, all message parts have failed the validation check. All measures were discarded. The body of the response contains detailed information.
401 Unauthorized	Authentication has failed, or the authenticated device is not authorized to post measures on behalf of the device with the alternate ID specified in the URL. For more information, please refer to section <a href="#">Device Authorization and Authentication [page 49]</a> .
404 Not Found	The requested URL refers to a non-existent REST endpoint.
405 Method Not Allowed	Only HTTP POST can be used for sending the measures.
500 Internal Server Error	The Internet of Things Gateway is not configured correctly, or a software error has occurred in the Internet of Things Gateway.
504 Gateway Timeout	This error is not returned by the Internet of Things Gateway itself, but rather by an HTTP proxy or reverse proxy located between the device and the Internet of Things Gateway. The most likely reason is that the Internet of Things Gateway is overloaded with requests, or that the measure message posted is very large and takes too long to process.

## 4.6.2.2 Response Body

When the device posts measures in the JSON format, it receives a detailed status response which is also JSON-formatted. The device must check the Content-Type HTTP header because a different content type may be returned by an HTTP proxy, or no response body at all may be returned in some cases.

The detailed status message is a list of objects. The number of objects is either one or corresponds to the number of measure messages posted within [Batched Measure Messages \[page 22\]](#). In the latter case, the

order of the objects is the same as the order of the measure messages in the batch. Each object can have the following fields:

```
{
  "code": "the status code for this Measure Message; the same as the HTTP status
  code if there is only one Measure Message in the HTTP request",
  "sensorAlternateId": "value from the request",
  "capabilityAlternateId": "value from the request",
  "messages": [a list of human-readable strings, mostly error messages, generated
  during the verification of the measure message]
}
```

Subsequent versions of the Internet of Things Gateway may extend the object with additional fields. All fields except for the `code` field can be missing (for example, if the JSON request could not be parsed, or no errors were detected).

The semantics of the `code` field is the same as of the corresponding HTTP status code (see the following table).

The `messages` field can contain any human-readable error messages. There is no fixed list of allowed values, but the messages may start with the strings listed in the table below.

Typical error messages are described in the following table.

Message	Condition
Unauthorized	The client is not authorized to post the measure messages on behalf of the device. In particular, this may mean an HTTP authentication failure.
Invalid device alternate ID	Most likely to appear when a valid and authenticated router device posts measure messages on behalf of another one.
Invalid message format	This means that the whole payload could not be parsed, for example due to invalid JSON syntax. The sensor and device alternate IDs, etc., cannot be extracted in this case.
Invalid sensor alternate ID	The sensor does not exist and could not be onboarded automatically.
Invalid capability alternate ID	The capability does not exist for the sensor, or it does not accept measures (being of the command type).
Invalid property alternate ID	The property does not exist in the capability.
Illegal property value	The value is not acceptable for the reason specified in the same error message. The message can be used if the measure is posted as a list with a wrong number of elements. It can also be used when the value for a certain property cannot be converted to the corresponding type.
No valid measurement data found	No property of the capability was assigned a valid value.

Message	Condition
Illegal time stamp value	The time stamp provided by the client for the measure message could not be parsed.
Gateway configuration error	Typically returned for a wrongly configured Internet of Things Gateway Edge. However, it may also be sent if there are errors in the data model, such as a capability without any properties.
Gateway internal error	Some other error was detected, and an internal consistency check has failed in the Internet of Things Gateway.

## 4.6.3 Measure Payload Customization

It is possible to use custom measure message format to inject measures into the Internet of Things Edge Platform REST.

### Prerequisites

The custom payload must be in JSON format.

### How to Enable

An apache freemarker template must be provided to transform the custom payload into a payload that respects the standard measure format. This will enable the Internet of Things Edge Platform to ingest custom payloads.

The template must be named `measureTemplate.ftl` and placed under the `config/freemarker` directory in the gateway folder.

### Example

To run this example, it is necessary to have:

A capability with the alternate ID `temperature_capability`.

The capability contains one property: Temperature of type float.

A sensor type with alternate ID 345 with the previously mentioned capability assigned with type measure.

Starting from the following custom payload:

#### Sample Code

```
{
  "time": 1604509375000,
  "sensor": "my-sensor",
  "value": 11.3
}
```

You can transform it into a standard measure by applying the following sample template:

#### Sample Code

```
<#escape output as output?json_string> (1)
{
  "sensorAlternateId": "${data.sensor}",
  "sensorTypeAlternateId": "345", (3)
  "capabilityAlternateId": "temperature_capability", (3)
  <#if data.time??> (2)
  "timestamp": #{data.time},
  </#if>
  "measures": [
    {
      "Temperature": #{data.value}
    }
  ]
}
</#escape>
```

#### Note

1. Ensure json escape is applied to the output.
2. Add the information only if time attribute is present in the original payload.
3. Fixed value for `sensorTypeAlternateId` and `capabilityAlternateId`.

## Customize Output on REST Edge Platform

In a similar way, it is possible to customize the response format for the ingestion in the REST Edge Platform.

In this case, the input format is the json response described here and you can apply your freemarker template, that must be named `OutputResponse.ftl` and placed under the `config/freemarker` directory in the gateway folder, to transform it into your custom response message.

Here is an example, starting from this response:

#### Sample Code

```
[{
  "code": "202",
  "sensorAlternateId": "my-sensor",
  "capabilityAlternateId": "mycapability",
  "messages": ["OK"]
}]
```



Applying the following template:

#### Sample Code

```
<#escape output as output?json_string>
{
  "sensor": "${data[0].sensorAlternateId}",
  "success": ${((data[0].code?starts_with("2")))?string('true','false')}
}
</#escape>
```

The result will be:

#### Sample Code

```
{
  "sensor": "my-sensor", (1)
  "success": true (2)
}
```

1. `sensorAlternateId` has been transformed in `sensor`, like we used in the ingestion payload.
2. The status code has been transformed in a Boolean that confirms if the ingestion has been successful.

## External Tools

If you need to try your template, you can use the following online tester from Apache foundation:

<https://try.freemarker.apache.org/>

#### Note

In the `DataModel` area, put `data=<your json>`

You must select `angleBracket` in the Tag Syntax input.

You must select `legacy` in the Interpolation Syntax input.

## 4.6.4 Command Message Format (JSON)

After a device has been successfully registered and a certain sensor has been assigned, the device can start receiving commands that conform to a capability defined for the respective sensor type.

The schema of a valid command JSON payload looks as follows:

```
{
  "title": "IoT Gateway JSON Message Schema",
  "type": "object",
  "properties": {
    "sensorAlternateId": {
      "id": "sensorAlternateId",
```

```

        "type": "string",
        "required": true,
        "description": "alternate id of the sensor the measures belong to"
    },
    "capabilityAlternateId": {
        "id": "capabilityAlternateId",
        "type": "string",
        "required": true,
        "description": "alternate id of the capability the command conform to"
    },
    "command": {
        "id": "command",
        "type": "object",
        "description": "command that conform to the capability definition with
the given alternate id"
    }
}
}
}

```

The following is an example of a valid command message published by the Internet of Things Gateway. A device can consume the message, by subscribing to the commands topic as described in the section *Endpoint Structure*.

#### Sample Code

```

{
  "sensorAlternateId": "sensor2",
  "capabilityAlternateId": "capability2",
  "command": {
    "LED": "on",
    "Buzzer": "off",
    "Speed": 45
  }
}

```

The payload includes a message that conforms to a capability with the alternate ID `capability2`. The command is for the sensor with the alternate ID `sensor2`. The command contains three properties, namely `LED`, `Buzzer`, and `Speed`.

## 4.6.5 Measure Message Format (Protobuf)

You can use the Internet of Things Gateway REST to ingest measures into the system in a binary format by using Google Protocol Buffers (protobuf).

The ingestion is made in the same way and at the same address as for the JSON format. To specify the use of the protobuf format in the POST request, set the value of the `Content-Type` attribute to `application/x-protobuf`:

```
Content-Type: application/x-protobuf
```

The body should be a binary (bytes) representation of a protobuf message that is compliant with the format declared by the file `MeasureRequest.proto`, as follows:

```

syntax = "proto3";
import "google/protobuf/any.proto";
package gateway;
option java_package = "com.sap.iotservices.common.protobuf.gateway";

```

```

option java_outer_classname = "MeasureProtos";
message MeasureRequest {
  string capabilityAlternateId = 1;
  string sensorAlternateId = 2;
  string sensorTypeAlternateId = 3;
  int64 timestamp = 4;
  repeated Measure measures = 5;

  message Measure {
    repeated google.protobuf.Any values = 1;
  }
}

```

When you create a protocol buffers schema for an existing message type consider the following:

- Protocol buffers do not support field names that start with numeric values and contain non-alphanumeric characters (except for the character '\_'). Make sure that you name your message type fields accordingly.
- The protocol buffers field names, types, and positions must match the message type. The position can be different, but in that case the measures field must be built in a specific way. For more information, please refer to section [Measure Value \[page 43\]](#).
- Fields of type `google.protobuf.Any` are stored as binary fields. We recommend that you use flat data structures.

### 4.6.5.1 Measure Identifier

- `Field_id`: **capabilityAlternateId**
- Field format: string

This field represents an identifier for the measure that is referred to by a measurement report. It maps to the capability alternate ID, which univocally identifies a certain measurement source hosted on the node. The field has the form of a single string, if the payload reports a single measurement.

### 4.6.5.2 Measure Timestamp

- `Field_id`: **timestamp**
- Field format: Java timestamp (Unix timestamp + milliseconds, for example, 1447407509060).

This field reports the timestamp of the measurement collection time, that is, the time of generation of the measurement on the device. This value is referred to as the measurement `start_time`, while the time of arrival in the Internet of Things Gateway is also known as `arrival_time`. This field is optional. If omitted, the Internet of Things Gateway parsing logic sets the measurement `start_time` equal to the `arrival_time`.

### 4.6.5.3 Measure Value

- `Field_id`: **measures**

- Field format: sequence (array) of sequences of the `google.protobuf.Any` type

This field reports the measurement values. The field can have the form of

- an array with only one inner array with a value for each inner property, in case the payload reports a single measure
- An array with more inner arrays, each of which with a value for each inner property, in case the payload reports a multiple measure.

#### Note

In this case, one of the properties of the capability must be of the type `measureTimestamp` and contain a different timestamp (expressed as milliseconds from epoch) for each measure.

To represent a capability with multiple properties, they are included as several `google.protobuf.Any` values in the same measure object, which results in a `measures` field that consists of only one value of the type `Measure`. This field contains several `values` fields, one for each property to represent.

It is allowed to have a different field order within the `Any` value than the one defined in the capability. In this case, the `typeUrl` value of the `Any` type is used to recognize the correct field type. Therefore, be sure that `typeUrl` contains the name of the property as defined in the capability.

If `typeUrl` is empty, the same order as defined in the capability is used to resolve the fields.

On the other hand, several `measures` objects of the type `Measure` are used to represent a time series. Each of these `measures` objects contains a single `values` field with one value of the series.

### 4.6.5.4 Sensor Alternate ID

- `Field_id`: `sensorAlternateId`
- Field format: ASCII string

This field reports the alternate ID of the sensor to which the measurement refers. If omitted, the Internet of Things Gateway automatically associates the measurement to a default sensor with the alternate ID `00:00:00:01`.

### 4.6.5.5 SensorType Alternate ID

- `Field_id`: `sensorTypeAlternateId`
- Required: **NO**
- Field format: integer

This field reports the `sensorTypeAlternateId` to which the measurement refers. If omitted, the gateway automatically associates the measurement to the default `sensorType` for the specific protocol (for example, 0 for CoAP, MQTT, REST). The field must always be in the form of a single, scalar number. Thus, the related `sensorTypeAlternateId` applies to the entire set of measurements reported in the current payload.

## 4.6.6 Report the Measure Processing Results (Protobuf)

The Internet of Things Gateway reports the processing results for the posted measures to the device using the REST API. The general outcome is reported as the status code returned for the corresponding HTTP request. The details including the error messages are returned in the body of the HTTP response which has the same content type as the request: if the measures are posted in the Google Protocol Buffers format, the detailed processing results are also returned in protobuf. If the measures are posted in the JSON format, the results are returned in JSON. For more information, please refer to section [Report the Measure Processing Results \(JSON\) \[page 39\]](#).

Note that the Internet of Things Gateway reports the processing status immediately, before forwarding the measures to the Internet of Things Service. If the Internet of Things Gateway considers the measure message invalid and rejects it, this outcome is final. However, if the Internet of Things Gateway accepts the measure message as valid, it may still be rejected or lost further down the road. Thus, the Internet of Things Gateway typically does not return the 200 `OK` status code, but only 202 `Accepted`. The REST API will be extended in future releases of the gateway, allowing HTTP clients to query the final processing outcome of measure messages posted earlier, including the processing by the Internet of Things Service.

### 4.6.6.1 HTTP Status Codes

The device must be able to handle at least the status codes described in the following table. This list is not exhaustive. Future versions of the Internet of Things Gateway may return additional status codes.

Status code	Meaning
200 <code>OK</code>	The Internet of Things Gateway has received a confirmation from the Internet of Things Service that the measure message was processed correctly before returning the result to the device. Almost never occurs.
202 <code>Accepted</code>	The measure message has passed the validation check executed by the Internet of Things Gateway.
207 <code>Multi-Status</code>	<a href="#">Batched Measure Messages [page 22]</a> were posted to the Internet of Things Gateway. Some messages in the batch have failed the validation, but at least one was accepted. The body of the response contains detailed information.
400 <code>Bad Request</code>	The measure message has failed the validation check executed by the Internet of Things Gateway. If the message posted is a batched message, all message parts have failed the validation check. All measures were discarded. The body of the response contains detailed information.

Status code	Meaning
401 Unauthorized	Authentication has failed, or the authenticated device is not authorized to post measures on behalf of the device with the alternate ID specified in the URL. For more information, please refer to section <a href="#">Device Authorization and Authentication [page 49]</a> .
404 Not Found	The requested URL refers to a non-existent REST endpoint.
405 Method Not Allowed	Only HTTP POST can be used for sending the measures.
500 Internal Server Error	The Internet of Things Gateway is not configured correctly, or a software error has occurred in the Internet of Things Gateway.
504 Gateway Timeout	This error is not returned by the Internet of Things Gateway itself, but rather by an HTTP proxy or reverse proxy located between the device and the Internet of Things Gateway. The most likely reason is that the Internet of Things Gateway is overloaded with requests, or that the measure message posted is very large and takes too long to process.

## 4.6.6.2 Response Body

When the device posts measures in the protobuf format, it receives a detailed status response which is also in the protobuf format. The device must check the Content-Type HTTP header because a different content type may be returned by an HTTP proxy, or no response body at all may be returned in some cases.

The definition of the corresponding type in the proto3 syntax is provided in the following sample. All fields are optional. More fields can be added in future versions of the Internet of Things Gateway.

```
message MeasureResponse {
  int32 code = 1; // the same as the HTTP status code
  string id = 2; // not returned by the current version
  int64 timestamp = 3; // not returned by the current version
  string sensorAlternateId = 4; // value from the request
  string capabilityAlternateId = 5; // value from the request
  repeated string messages = 6; // a list of human-readable strings,
                                // mostly error messages, generated during
                                // the verification of the measure message
}
```

The current version of the Internet of Things Gateway does not support the protobuf format for batched measure messages. Therefore, neither individual identifiers nor individual timestamps for measure messages are needed.

The `messages` field can contain any human-readable error messages. There is no fixed list of allowed values, but the messages may start with the strings listed in the following table.

Typical error messages are described in the following table.

Message	Condition
Unauthorized	The client is not authorized to post the measure messages on behalf of the device. In particular, this may mean an HTTP authentication failure.
Invalid device alternate ID	Most likely to appear when a valid and authenticated router device posts measure messages on behalf of another one.
Invalid message format	This means that the whole payload could not be parsed, for example due to invalid JSON syntax. The sensor and device alternate IDs, etc., cannot be extracted in this case.
Invalid sensor alternate ID	The sensor does not exist and could not be onboarded automatically.
Invalid capability alternate ID	The capability does not exist for the sensor, or it does not accept measures (being of the command type).
Invalid property alternate ID	The property does not exist in the capability.
Illegal property value	The value is not acceptable for the reason specified in the same error message. The message can be used if the measure is posted as a list with a wrong number of elements. It can also be used when the value for a certain property cannot be converted to the corresponding type.
No valid measurement data found	No property of the capability was assigned a valid value.
Illegal time stamp value	The time stamp provided by the client for the measure message could not be parsed.
Gateway configuration error	Typically returned for a wrongly configured Internet of Things Gateway Edge. However, it may also be sent if there are errors in the data model, such as a capability without any properties.
Gateway internal error	Some other error was detected, and an internal consistency check has failed in the Internet of Things Gateway.

## 4.6.7 Command Message Format (Protobuf)

After a device has been successfully registered and a certain sensor has been assigned, the device can start receiving commands that conform to a capability defined for the respective sensor type.

It is possible to send commands as binary data by using Google Protocol Buffers (protobuf) encoding as payload for command messages. The format is shown in the following `CommandResponse.proto` file:

```
syntax = "proto3";
```

```

import "google/protobuf/any.proto";
package gateway;
option java_package = "com.sap.iotservices.common.protobuf.gateway";
option java_outer_classname = "CommandResponseProtos";
message CommandResponse {
    string capabilityAlternateId = 1;
    string sensorAlternateId = 2;
    Command command = 3;

    message Command {
        repeated google.protobuf.Any values = 1;
    }
}

```

The command field is of type `Command`, which is defined as a sequence (array) of type `google.protobuf.Any` with the purpose to match the properties of the capability that represents the command specified with the capability alternate ID.

In case multiple commands are delivered to the device, they are included in a `CommandList` message, so that they are delivered as an array of command objects. The following `CommandResponseList.proto` file contains the syntax:

```

syntax = "proto3";
import "CommandResponse.proto";
package gateway;
option java_package = "com.sap.iotservices.common.protobuf.gateway";
option java_outer_classname = "CommandResponseListProtos";
message CommandResponseList {
    repeated CommandResponse commands = 1;
}

```

When you create a protocol buffers schema for an existing message type consider the following:

- Protocol buffers do not support field names that start with numeric values and contain non-alphanumeric characters (except for the character '\_'). Make sure that you name your message type fields accordingly.

## 4.6.8 Device Authorization and Authentication

The `alternateId` of the sending device is part of the URL path. In each client connection, the following authorization checks are performed:

### Secure Transport

- The `instanceId` included in the certificate must match the ID of the instance to which the Internet of Things Gateway belongs.
- The `deviceAlternateId` included in the certificate must match the `deviceAlternateId` used in the URL path.

The client authorization logic cross-checks the metadata provided by the connecting device through its client certificate with the resources the connecting device is trying to access.



## 4.6.9 Endpoint Structure

### 4.6.9.1 Secure Transport

The default port of the Internet of Things Edge Platform REST is 8699. Measures can be ingested into the Internet of Things Edge Platform REST by performing a POST call on the URL:

```
https://<IOT_GATEWAY_IP>:8699/measures/<deviceAlternateId>
```

The `deviceAlternateId` represents the alternate ID of the device that generated the measures.

Commands can be retrieved from the Internet of Things Edge Platform REST by performing a GET call on the following URL:

```
https://<IOT_GATEWAY_IP>:8699/commands/<deviceAlternateId>
```

The specific configuration of the REST service for data ingestion exposed by Internet of Things Edge Platform REST relies on the following configuration files, `sap.NettyConfiguration.cfg` and `sap.RestAdapterConfiguration.cfg` under the folder `<GatewayEdge_deploy_folder>/config/services`.

The file `sap.NettyConfiguration.cfg` has the following standard format:

```
# Enables a Netty server
enabled = true
# Host name or IP address to identify a specific network interface on which to
listen
host = localhost
# TCP/IP port on which the server listens for connections
port = 8699
# Expose service on https
enableSSL = false
```

You must change the value for the property `enableSSL` to `true` to expose the service with HTTPS.

The file `sap.RestAdapterConfiguration.cfg` has the following standard format:

```
# Enable/Disable device authentication based on ssl_client_user header
useDeviceAuth = true
```

The property `useDeviceAuth` must be kept to `true` to cross-check the metadata provided by the connecting device through its client certificate with the resources the connecting device is trying to access.

The supported service configurations are listed below:

1. `enableSSL = false` and `useDeviceAuth = false`: The REST server is exposed via plain, unsecure HTTP.
2. `enableSSL = true` and `useDeviceAuth = true`: The REST server is exposed via HTTPS with mutual authentication, and enforces a check on the certificate which the clients must provide.

#### Note

Please refer to the section "Send data to Internet of Things Edge Platform REST with the secure transport using cURL" in .

The following steps are required to run Internet of Things Edge Platform REST with the secure transport enabled.

1. Configure the HTTPS endpoint in the configuration file `sap.NettyConfiguration.cfg`

```
# Enables a Netty server
enabled = true
# Host name or IP address to identify a specific network interface on which
to listen
host = localhost
# TCP/IP port on which the server listens for connections
port = 8699
# Expose service on https
enableSSL = true
```

2. Start the Internet of Things Edge Platform REST by launching `gateway.bat / .sh`

## 4.6.9.1.1 Custom Certificates

### Overview

It is possible to change the REST Edge Platform default behavior for publishing the HTTPS endpoints using a custom certificate, instead of relying on the SAP IoT certificates.

In this way, the customers can use their own certificates to expose the HTTPS endpoints using, for example, a certificate signed from a trusted certificate authority.

### How to Enable

To enable this feature, the REST Edge Platform must be configured in the following way:

- Prepare a `.p12` file with the custom certificate to be used for publishing the endpoints.
- Store it into the Internet of Things Edge Platform file system, under the folder `'config/certificates/server'` (if the folder does not exist, create it).
- Go to the `'sap.NettyConfiguration.cfg'` config file (under the folder `'config/services'`) and:
  - Enable the secure transport to expose the service with HTTPS, setting the property `enableSSL` to `true`.
  - Enable the usage of the custom certificate, setting the property `useCustomServerCertificate` to `true`.
  - Configure the path to the custom certificate, setting the property `customKeyStoreFilePath`.
  - Configure the secret required to open the custom certificate, in one of these two ways:
    - Setting the secret via the property `customKeyStoreSecret`.
    - Passing the secret to the Internet of Things Edge Platform via a System Variable, and setting the name of that variable via the property `customKeyStoreSecretEnvVariable`.

Given below is a `'sap.NettyConfiguration.cfg'` file template, specifying the secret directly into the file:

### Sample Code

```
# Enables a Netty server
enabled = true
# Host name or IP address to identify a specific network interface on which
to listen
host = localhost
# TCP/IP port on which the server listens for connections
port = 8699
# Expose service on https
enableSSL = true
# Enable usage of a custom certificate
useCustomServerCertificate = true
# Sub path where the custom certificate is stored
customKeyStoreFilePath = config/certificates/server/my-certificate.p12
# Secret for custom certificate
customKeyStoreSecret = p6N9NUm87MxGUvm9
```

Here is a 'sap.NettyConfiguration.cfg' file template, passing the secret via System Variable:

### Sample Code

```
# Enables a Netty server
enabled = true
# Host name or IP address to identify a specific network interface on which
to listen
host = localhost
# TCP/IP port on which the server listens for connections
port = 8699
# Expose service on https
enableSSL = true
# Enable usage of a custom certificate
useCustomServerCertificate = true
# Sub path where the custom certificate is stored
customKeyStoreFilePath = config/certificates/server/my-certificate.p12
# Name of the system variable containing the secret for custom certificate
customKeyStoreSecretEnvVariable = myEnvVariableForSecret
```

## 4.6.9.1.1.1 Prerequisites

Please follow the below prerequisites to ensure that the feature works properly and avoid any unexpected behavior.

### Certificate's Format and Secret

- The certificate must be stored as .p12 file (other formats are not supported).
- The secret to open the .p12 file cannot be empty.

## Certificate's Health and Security Checks

The Internet of Things Edge Platform reserves the right to validate the certificate before using it; in case it does not satisfy the security checks, the publication of the HTTPS endPoints will fail.

Here are the main requirements for trusted certificates:

- TLS server certificates and issuing CAs using RSA keys must use key sizes greater than or equal to 2048 bits. Certificates using RSA key sizes smaller than 2048 bits are not valid.
- TLS server certificates and issuing CAs must use a hash algorithm from the SHA-2 family in the signature algorithm. SHA-1 signed certificates are not valid.
- TLS server certificates must not be expired.

## Certificate Renewal

### Note

The customer is responsible for the certificate renewal, and must take care of configuring a new custom certificate on the Internet of Things Edge Platform, before the previous one is expired. A restart of the REST endPoints will be required to make the renewal effective.

Here are the main steps for renewing a certificate:

- Store the new certificate in the Internet of Things Edge Platform file system, under the folder `'config/certificates/server'`.
- Change the `'sap.NettyConfiguration.cfg'` file (under the folder `'config/services'`), updating the field `'customKeyStoreFilePath'` to point to the new file (as a result of this, the Internet of Things Edge Platform will automatically restart the REST endPoints).
- Update the certificate secret, if changed (setting the value of the new secret via the config file or updating the value of the system variable, as described before).

## 4.6.9.1.1.2 Device Authorization

Once the REST Edge Platform runs in the HTTPS mode, by default, it publishes the REST endPoints requiring the mutual auth (that is, the client certificate is mandatory) and using the client certificate metadata to check if the device is authorized to connect or not. As a result, a device can be authorized only if it uses a certificate released by SAP IoT service. For this reason, to keep the device authorization enabled, the devices must continue using SAP IoT certificates, even if the Internet of Things Edge Platform is configured for publishing the HTTPS endpoints with a custom certificate.

If the customer is willing to disable the device authorization and rely only on the device authentication given by the usage of a trusted certificate, the customer can configure the Internet of Things Edge Platform to accept HTTPS connection from devices using a custom certificate.

The main steps to do this are:

- Go to the 'sap.RestAdapterConfiguration.cfg' config file (under the folder 'config/services') and disable the device authorization checks, setting the parameter useDeviceAuth to false. Below is a 'sap.RestAdapterConfiguration.cfg' file template:

#### Sample Code

```
# Enable/Disable the device authorization checks
useDeviceAuth = false
```

- If the device certificate is signed from a custom CA, that CA must be added to the Internet of Things Edge Platform truststore as per the below steps:
  - Prepare a .p12 file containing the required CAs.
  - Store it into the Internet of Things Edge Platform file system, under the folder 'config/certificates/server'.
  - Go to the 'sap.NettyConfiguration.cfg' config file (under the folder 'config/services') and:
    - Configure the path to the custom CA, setting the property customTrustStoreFilePath.
    - Configure the secret required to open the custom CA, in one of these two ways:
      - Setting the secret via the property customTrustStoreSecret.
      - Passing the secret to the Internet of Things Edge Platform via a System Variable, and setting the name of that variable via the property customTrustStoreSecretEnvVariable.

Given below is a 'sap.NettyConfiguration.cfg' file template, specifying the CA secret directly into the file:

#### Sample Code

```
# Enables a Netty server
enabled = true
# Host name or IP address to identify a specific network interface on which
to listen
host = localhost
# TCP/IP port on which the server listens for connections
port = 8699
# Expose service on https
enableSSL = true
# Enable usage of a custom certificate
useCustomServerCertificate = true
# Sub path where the custom certificate is stored
customKeyStoreFilePath = config/certificates/server/my-certificate.p12
# Secret for custom certificate
customKeyStoreSecret = p6N9NUm87MxGUvm9
# Sub path where the custom CA is stored
customTrustStoreFilePath = config/certificates/server/my-ca.p12
# Secret for custom CA
customTrustStoreSecret = caSecret!
```

Given below is a 'sap.NettyConfiguration.cfg' file template, passing the secret via System Variable:

#### Sample Code

```
# Enables a Netty server
enabled = true
# Host name or IP address to identify a specific network interface on which
to listen
host = localhost
# TCP/IP port on which the server listens for connections
port = 8699
# Expose service on https
```

```
enableSSL = true
# Enable usage of a custom certificate
useCustomServerCertificate = true
# Sub path where the custom certificate is stored
customKeyStoreFilePath = config/certificates/server/my-certificate.p12
# Name of the system variable containing the secret for custom certificate
customKeyStoreSecretEnvVariable = myEnvVariableForSecret
# Sub path where the custom CA is stored
customTrustStoreFilePath = config/certificates/server/my-ca.p12
# Name of the system variable containing the secret for custom CA
customTrustStoreSecretEnvVariable = customTrustStoreSecret
```

## Note

In case a custom CA is configured:

- Its public key must be stored into a .p12 file (other formats are not supported).
- The secret required to open the .p12 file cannot be empty.
- The public key contained in the .p12 file must satisfy the same security requirements as mentioned in the section "Certificate's Health and Security Checks" in [Prerequisites \[page 107\]](#).
- The customer is responsible for the CA renewal and must take care of configuring a new CA on the Internet of Things Edge Platform, before the previous one is expired. A restart of the REST endPoints will be required to make the renewal effective.

The main steps for renewing a CA are listed below:

- Store the new CA file in the Internet of Things Edge Platform file system, under the folder 'config/certificates/server'.
- Change the 'sap.NettyConfiguration.cfg' file (under the folder 'config/services'), updating the field 'customTrustStoreFilePath' to point to the new file (as a result of this, the Internet of Things Edge Platform will automatically restart the REST endPoints).
- Update the certificate secret, if changed (setting the value of the new secret via the config file or updating the value of the system variable, as described before).

## 4.6.9.2 Insecure Transport

The default port of the Internet of Things Edge Platform REST is 8699. Measures can be ingested into the Internet of Things Edge Platform REST by performing a POST call on the URL:

```
http://<IOT_GATEWAY_IP>:8699/measures/<deviceAlternateId>
```

The `deviceAlternateId` represents the alternate ID of the device that generated the measures.

Commands can be retrieved from the Internet of Things Edge Platform REST by performing a GET call on the following URL:

```
http://<IOT_GATEWAY_IP>:8699/commands/<deviceAlternateId>
```

## 4.7 CoAP

CoAP (Constrained Application Protocol) is intended for use with constrained nodes and networks.

### ⓘ Note

The CoAP adapter is only available for the Internet of Things Edge Platform. For more information, please refer to the tutorial [Set Up the Internet of Things Edge Platform - CoAP](#).

### Related Information

[Overview \[page 111\]](#)

[Adapter Configuration \[page 111\]](#)

### 4.7.1 Overview

The data injection for the Internet of Things Edge Platform CoAP is based on the following paradigm:

- Device: acting as CoAP client
- Internet of Things Edge Platform: hosting a CoAP server

By default, the Internet of Things Edge Platform hosts a CoAP resource named `measures`, which can be used to push data from devices by means of CoAP `POST` or `PUT` calls.

### ⓘ Note

Only communication from the CoAP client to the Internet of Things Edge Platform is supported. Communication from the Internet of Things Edge Platform to the CoAP client, in form of generic commands for example, is not supported.

### 4.7.2 Adapter Configuration

To configure the Internet of Things Edge Platform CoAP, you have to edit two files with different settings.

The Internet of Things Edge Platform hosts a CoAP server that devices can push data to through `POST` or `PUT` calls. The default COAP resource to be targeted by the requests is named 'measures'.

#### **config/config\_gateway\_coap.xml**

First, edit the `config_gateway_coap.xml` file in the `config` folder.

Open the sections **Connection to the Internet of Things Messaging** and **Connection to the Internet of Things Service** to learn more about the configuration steps that are common to every Internet of Things Edge Platform adapter.

## Connection to the Internet of Things Messaging

The connection to the Internet of Things Messaging is controlled by the first `<cnf:amq>` node in the configuration file of the Internet of Things Edge Platform. The `cnf:connectionString` node controls the connection of the Internet of Things Edge Platform to the Internet of Things Messaging. It must be specified in the form

```
failover:(nio+ssl://<hostname>:61616?daemon=true&soTimeout=60000)
```

The failover option allows the system to reconnect to the Internet of Things Messaging as soon as the network connectivity is restored after a disconnection. `<hostname>` represents the hostname of the instance that the Internet of Things Edge Platform connects to and is typically the only parameter that you have to change in the connection string. The `nio+ssl` section identifies the protocol to use for the connection. Currently, only the nio-ssl transport can be used, exposed on port 61616 of the production instances.

## Connection to the Internet of Things Service

The connection to the Internet of Things Service is controlled by the `<cnf:coreBundles>/<cnf:endpoints>` node. When connecting an Internet of Things Edge Platform to an Internet of Things Service, you have to set the following parameters within the nodes:

- `<cnf:cxf lan="false">/<cnf:address>`: insert the hostname of the instance to connect to.
- `<cnf:cxf lan="false">/<cnf:restPort>`: insert the service port. The default value for this field (443) matches the actual port where the Internet of Things Service exposes REST services, therefore, you don't have to change it.

### Note

The following configuration steps are specific for the Internet of Things Edge Platform COAP:

## Configuration of the Internet of Things Edge Platform

The items under the `cnf:gatewayBundle` node include all of the information affecting the internal configuration of the Internet of Things Edge Platform (that is, not relating to the communication with other components).

- Make sure that `gatewayver` key is set to COAP.

```
...  
<cnf:gatewayBundle gatewayver="COAP">  
...
```



- In the gateway configuration part, set the `technology` field to the value `coap`.

```
...
<cnf:gateway>
  <cnf:technology>coap</cnf:technology>
...
```

- The `cnf:gateway_comm` field allows you to control the IP address and the port that the CoAP server will bind to, to be set respectively through the parameters `cnf:socket_host` and `cnf:socket_port`. The default value of the port is 5683, which is the value assigned to COAP by the Internet Assigned Numbers Authority (IANA).

Set `cnf:media` to `COAP` and `cnf:useAsServer` to `true`.

```
<cnf:gateway_comm>
  <cnf:media>COAP</cnf:media>
  <cnf:useAsServer>true</cnf:useAsServer>
  <cnf:socket_host>127.0.0.1</cnf:socket_host>
  <cnf:socket_port>5683</cnf:socket_port><!--7710-->
</cnf:gateway_comm>
```

- Set the field `useLocal` under `dataModel` to `true`.

```
<cnf:dataModel>
  <cnf:useLocal>true</cnf:useLocal>
</cnf:dataModel>
```

## config/schemas

When the Internet of Things Gateway CoAP adapter starts up, it searches for schema files to use when parsing an incoming message. These files are in JSON format and collected in `config/schemas` folder.

### Note

To run the adapter, at least the file containing the default schema, named `default_payload_COAP.json`, has to be present in the `schema` folder.

## Run

To run the Internet of Things Edge Platform CoAP, execute the script `./gateway.sh`. The console output is displayed.

## 4.8 FILE

The FILE adapter targets the retrieval of data from files available on the file system of the machine where the Internet of Things Edge Platform FILE is running.

### Note

The FILE adapter is only available for the Internet of Things Edge Platform. For more information, please refer to the tutorial [Set Up the Internet of Things Edge Platform - FILE](#).

### Related Information

[Overview \[page 114\]](#)

[Adapter Configuration \[page 114\]](#)

### 4.8.1 Overview

The typical workflow of the adapter is the following:

1. Read a certain file, based on a set of configuration parameters.
2. Go through the file content and extract valuable fields as instructed by an adapter parsing schema file.
3. Map such fields onto the concepts and entities of the Internet of Things Service (for example, device addressing metadata, measurements and so on), also based on the parsing schema file of the adapter.
4. Stream normalized data up to Internet of Things Service.
5. Remove the target file from its original location and, optionally, back it up into another folder.

The supported file formats are JSON, CSV, and binary. For binary, the file is transferred as a Base64 encoded string to the Internet of Things Service.

### 4.8.2 Adapter Configuration

When the Internet of Things Edge Platform File adapter starts up, it searches for the following configuration material:

- The main XML configuration file: `config_gateway_file.xml`.
- An Internet of Things Service adapter specific configuration file, `config/readFileConfig/readFile.json`. This file contains data regarding the set of files to analyze.
- A set of schema files used as syntax definition at the moment of parsing incoming messages. These files are in JSON format and are stored in the `config/schemas` folder. Before running the adapter, make sure that at least the files with the schemas in the file `readFile.json` are present in the `schema` folder. The default deployment for the Internet of Things Edge Platform File adapter comes with

three pre-configured example schema files: `BinaryFileSchema.json`, `PeopleCountSchema.json`, and `WeatherStationSchema.json`.

The `readFile.json` file determines the interaction of the Internet of Things Edge Platform File with its target files in the local file system. It is written in JSON format and contains a root object named `files`, whose value is an array of items with an entry for each single file to monitor.

For each object entry, the following fields should be specified:

- `path`: the path to the file to be analyzed, either as an absolute path or as a relative path. By specifying a directory as a target, all files included into this directory are analyzed
- `mode`: the way the file should be read:
  - `periodic`: read the file whenever a timeout is expired
- `interval`: timeout period in milliseconds for periodic mode
- `schema`: the name of the schema describing the file content to use. This should match a file named `<schemaName>.json` placed inside `schemas` folder under `config`
- `deviceAlternateId`: the identifier of the device (`deviceAlternateId`), mapped to the input file. If the device does not exist in the system yet, it is automatically created by the Internet of Things Edge Platform
- `backupFolder`: a path to the folder where to store the backup of already analyzed files

## 4.9 Modbus

Modbus defines an application layer messaging protocol for client/server communication between devices connected on different types of buses or networks.

### Note

The Modbus adapter is only available for the Internet of Things Edge Platform. For more information, please refer to the tutorial [Set Up the Internet of Things Edge Platform - Modbus](#).

## Related Information

[Overview \[page 116\]](#)

[Adapter Configuration \[page 117\]](#)

[Modbus Function Codes \[page 120\]](#)

[Node and Data Item Addressing \[page 120\]](#)

## 4.9.1 Overview

Modbus is currently implemented using:

- TCP/IP over Ethernet
- Asynchronous serial transmission over various media (wire: RS-232, RS-485; fiber, radio, and so on)
- ModbusPLUS, a high-speed token passing network.

Modbus defines a simple protocol data unit that is independent of the underlying communication layers. The mapping on specific buses or networks can introduce some additional fields on the application data unit. The protocol follows a client-server approach, where a client initiates a Modbus transaction to poll a server for data.

The protocol data model is based on a series of tables that have different characteristics. The four primary tables are summarized in the following:

Primary table	Object type	Access type
Discrete input	Single bit	Read-only
Coil	Single bit	Read-write
Input register	16-bit word	Read-only
Holding register	16-bit word	Read-write

The distinctions between inputs and outputs, and between bit-addressable and word-addressable data items, do not imply any application behavior. All four tables can be regarded as overlaying one another. For each of the primary tables, the protocol allows the individual selection of 65536 data items. The operations of read or write of those items are designed to span multiple consecutive data items. The reference numbers used to address a data item are unsigned integer indexes starting at zero.

### Note

Modbus is data-agnostic, in the sense that it does not specify what type of information is contained in a certain item. For example, Modbus does not specify whether the data represents a temperature or a current value.

## Requirements

The Internet of Things Edge Platform Modbus adapter meets the following requirements:

- Compliancy with Modbus over TCP/IP protocol
- The retrieval of data from multiple devices on a TCP network
- The selection of the data items that are represented in the Internet of Things Service among the entirety of items available on the device
- Flexibility in the addressing scheme used to reference the data in the Internet of Things Service

In the context of the Modbus client/server architecture, the Modbus adapter of the Internet of Things Edge Platform implements a client (a Modbus master) that is able to poll a set of servers (Modbus masters).

Servers are represented as devices in the Device Management API. For each server, the Internet of Things Edge Platform adapter retrieves values from a configured set of registers, and performs measurement ingestion to the Internet of Things Service accordingly. The current version of the Internet of Things Edge Platform Modbus adapter only supports the ingestion of measurements, for example the reading of server registers. Writing a server register through a device command is not supported.

## 4.9.2 Adapter Configuration

To configure the Internet of Things Edge Platform Modbus adapter, you have to edit an XML and a JSON file with different settings.

### config/config\_gateway\_modbus.xml

First, edit the `config_gateway_modbus.xml` file in the `config` folder.

Open the sections **Connection to the Internet of Things Messaging**, **Connection to the Internet of Things Service**, and **Configuration of Internet of Things Edge Platform** to learn more about the configuration steps that are common to every Internet of Things Edge Platform adapter.

### Connection to the Internet of Things Messaging

The connection to the Internet of Things Messaging is controlled by the first `<cnf:amq>` node in the configuration file of the Internet of Things Edge Platform. The `cnf:connectionString` node controls the connection of the Internet of Things Edge Platform to the Internet of Things Messaging. It must be specified in the form

```
failover:(nio+ssl://<hostname>:61616?daemon=true&soTimeout=60000)
```

The failover option allows the system to reconnect to the Internet of Things Messaging as soon as the network connectivity is restored after a disconnection. `<hostname>` represents the hostname of the instance that the Internet of Things Edge Platform connects to and is typically the only parameter that you have to change in the connection string. The `nio+ssl` section identifies the protocol to use for the connection. Currently, only the `nio-ssl` transport can be used, exposed on port 61616 of the production instances.

### Connection to the Internet of Things Service

The connection to the Internet of Things Service is controlled by the `<cnf:coreBundles>/<cnf:endpoints>` node. When connecting an Internet of Things Edge Platform to an Internet of Things Service, you have to set the following parameters within the nodes:

- `<cnf:cxf lan="false">/<cnf:address>`: insert the hostname of the instance to connect to.

- `<cnf:cxflan="false">/<cnf:restPort>`: insert the service port. The default value for this field (443) matches the actual port where the Internet of Things Service exposes REST services, therefore, you don't have to change it.

## Configuration of the Internet of Things Edge Platform

As part of the bootup procedure, an Internet of Things Edge Platform presents an identification string called `gatewayAlternateId`, which is sent to the Internet of Things Service and validated to either allow or prevent the startup of the Internet of Things Edge Platform to complete. In general, you can retrieve the `gatewayAlternateId` from the configuration file, where it can be provided as an attribute of the `cnf:gatewayBundle/cnf:gateway` node:

```
<cnf:gateway gatewayAlternateId="test-network">
```

The `gatewayAlternateId` attribute must be a valid ASCII String.

### Note

The following configuration steps are specific for the Internet of Things Edge Platform Modbus:

## config/modbusCfg/config.json

The subfolder `modbusCfg` of the `config` folder contains rest-specific configuration files. Edit the `generic.json` file in the `modbusCfg` folder to set up the REST server.

The file contains – in JSON format – the list of devices and the format of the measure they manage. There is a `devices` section, containing a list of data to access a certain Modbus server. First, some access information is provided:

- `ip`: Modbus server IP address
- `port`: Modbus server TCP port
- `unitid`: Unit ID: the use of this value depends on the specificities of the addressed Modbus device. Many vendors have developed gateway products, which receive a Modbus TCP packet, look at the internal Slave ID then they reformulate a new Modbus RTU packet that is sent out the serial port of the gateway to the intended target device. In such cases, the Unit ID is used to point to the correct slave device acodephached to the Modbus server, so its value shall match the correct slave ID. On the other hand, in purely TCP/IP terms the IP address of a Modbus TCP/IP device could be considered the actual address of the slave; so, when dealing with stand-alone Modbus devices (for example, not acting as gateways to a RTU bus), the Unit ID should normally be set to 0 or 1.
- `regoffset`: Register offset: used to add a fixed offset to the Modbus reading/writing functions. As an example, reading register 4 with offset 100 will result in register 104 being read. Normally, this value is kept equal to 0 and the real register value (with respect to 0) is provided.
- `deviceAlternateId`: This value is used as `alternateId` for the device associated to this Modbus server. As the device `alternateId` must be unique across the devices connecting the same gateway, each server in this configuration file must be assigned a unique value (derived for example, from a MAC address, a serial number or a custom-defined value).

The following is an extract of `config.json` containing the definition of one Modbus server.

```
"devices":
  [ {
    "ip": "10.0.3.59",
    "port": 502,
    "unitid": 1,
    "regoffset": 0,
    "deviceAlternateId": "00:ae:23:01:33:f2:00:01",
    ...
  }
```

The `measures` section defines how to map Modbus register into measurements, each associated to the usual set of identifiers (`sensorTypeAlternateId`, `capabilityAlternateId`) and some other configuration parameters.

- `polling`: Wait time (in milliseconds) between consecutive Modbus operations on the same register
- `regaddr`: Starting register for the Modbus operation
- `regcount`: Number of register addressed by the Modbus operation
- `functionmod`: Modbus function code of the operation to be performed on the set of registers specified by `regaddr`, `regcount`
- `sensorTypeAlternateId`: the alternate ID of the corresponding sensor type (profile)
- `capabilityAlternateId`: The alternate ID onto which the data shall be mapped
- `reverseByte`: Set to `true` to swap the bit order in a certain byte. Normally set to `false`.
- `waitBeforeTrans`: If set to `true`, a certain wait time will be observed before each reading operation to the server. This interval is set to 2 seconds by default; it may be changed by modifying the custom property `waitBeforeTransaction` (for example, `waitBeforeTransaction=500` will set a 500 msec wait time. Default value is 2000)
- `reconnectBeforeTrans`: Parameter for the Modbus transaction. Forces an automatic reconnection before each transaction. Normally set to `false`.

```
...
"measures":
  [ {
    "polling": 20000,
    "regaddr": 0,
    "regcount": 2,
    "functionmod": 3,
    "sensorTypeAlternateId": 7,
    "capabilityAlternateId": 30,
    "reverseByte": false,
    "reverseEndianess": false,
    "waitBeforeTrans": true,
    "reconnectBeforeTrans": false
  }, ...
  ]
}
```

## Run

To run the Internet of Things Edge Platform Modbus, execute the script `./gateway.sh`. The console output is displayed.

## Test Environment

To test the Internet of Things Edge Platform Modbus adapter, you need an external tool acting as Modbus server, for example, Ananas. It can be configured to send measures both for Input or Holding registers. For more information, please refer to the tutorial [Set Up the Internet of Things Edge Platform - Modbus](#).

### 4.9.3 Modbus Function Codes

The Internet of Things Gateway supports the most common functions that servers provide for reading and writing registers, that is, the `Read holding registers` (function code = 3), the `Read input registers` (function code = 4), the `Write Single Coil` (function code = 5) and the `Write Single Register` (function code = 6).

## Write Commands

The Internet of Things Gateway Modbus Adapter provides two commands to allow writing on the Modbus Servers.

### Function Code 05 (0x05) Write Single Coil

The first command, `writeSingleCoil`, corresponds to the function code 5 and is used to write a single output to either **ON** or **OFF** in a remote device. It takes the below 2 parameters:

- `address` (Integer): The address of the Coil to set. Coils are addressed starting at zero; therefore coil numbered 1 is addressed as 0.
- `value` (Boolean): The requested **ON/OFF** state is specified by a boolean. A value of true requests the coil to be **ON**. A value of false requests the coil to be **OFF**.

### Function Code 06 (0x06) Write Single Register

The second command, `writeSingleRegister`, corresponds to the function code 6 and is used to write a single holding register in a remote device. It takes the below 2 parameters:

- `address` (Integer): The address of the Holding Register to set. Registers are addressed starting at zero; therefore register numbered 1 is addressed as 0.
- `value` (Integer). The short value to write in the register, from -32768 to 32767 corresponding to the hex values 0x0000 to 0xFFFF.

### 4.9.4 Node and Data Item Addressing

The addressing for a Modbus node must represent the following information:

- MAC/IP addresses of the device



- TCP port on which the Modbus server is listening
- The type of data item (one of the 4 basic types previously specified)
- The reference (address) of the data item
- The type of measurement contained (the physical measurement contained in the data item)

The addressing structure is the following:

- Physical address: the MAC address of the physical device
- Protocol address: the IP address of the device
- Sensor address: composed on the base of the TCP port, data item type, and data item address of the device. In particular:
  - bytes 0-1: data item address
  - byte 2: type of data item (0 for discrete input; 1 for coil; 2 for input register; 3 for holding register)
  - bytes 3-4: device TCP port
- objectID: maps the nature of the data contained at the specified address

## 4.10 OPC UA

The OPC UA (Open Platform Communication Unified Architecture) is a platform independent service-oriented architecture that has emerged as one of the most important standard in the field of industrial automation. The OPC Foundation is responsible for its development and maintenance. Initially, the OPC standard was restricted to the Windows operating system: this specification, known as OPC Classic, has reached wide adoption across multiple industries. In 2008, the OPC Foundation released the OPC UA specifications to address new challenges in security and data modelling, providing a technology based on an open-platform architecture that aims to be future-proof, scalable, and extensible.

### Note

The OPC UA adapter is only available for the Internet of Things Edge Platform. For more information, please refer to the tutorial [Set Up the Internet of Things Edge Platform - OPC UA](#).

## Related Information

[Overview \[page 121\]](#)

[Adapter Configuration \[page 123\]](#)

[Device Model Mapping \[page 126\]](#)

[Mapping Business Objects in the Internet of Things Edge Platform OPC UA \[page 128\]](#)

### 4.10.1 Overview

The Internet of Things Edge Platform integrates OPC UA servers by acting as an OPC UA client. The adapter can connect to the server to establish an OPC UA session. At startup, a device is created for each OPC

UA server configured with a sensor named `Commands` that allows running commands. For more information, please refer to section [Device Model Mapping \[page 126\]](#).

It is possible to configure the OPC UA gateway adapter to browse automatically the server and create all the sensors corresponding to variable nodes containing values. This option should be used only if there is a priori knowledge of the server and it is known it contains a limited number of nodes.

To configure this automatic onboard, the value of parameter `com.sap.iotservices.gatewayopcuaBuildAllOnStartup` defined in `gateway.bat` (Windows) or `gateway.sh` (macOS) must be set to `true`.

If the automatic onboard of all the nodes is not used (as recommended), it is possible to browse the server to receive the list of variable nodes (for example, nodes with values), using the command `BrowseNodes`.

The command `OnBoardNodes` is the one to be used to create a sensor that maps an OPC UA node to retrieve values. The retrieval of data can be configured with one of two possible options:

- `Periodic pull mode`: the adapter polls the server for data on a periodic base, with a configurable polling interval
- `Subscription mode`: the adapter creates a subscription on the server nodes, to be notified by the server itself whenever the value of a node changes

Additionally, the adapter is capable of exposing commands to set the value of an OPC UA node, or as well to create a new node in the server.

The commands defined are the following:

Name	Parameters	Description
<code>BrowseServer</code>	<ul style="list-style-type: none"> <li>• <code>serverId</code> (string): The server to browse.</li> </ul>	Performs a browse of the structure defined on the given <code>serverId</code> . As a result, a new measure value is sent for the capability <code>StringValue</code> of the sensor <code>Commands</code> . The value is a JSON formatted representation of the nodes defined by the server.
<code>OnboardNodes</code>	<ul style="list-style-type: none"> <li>• <code>jsonNodesToOnboard</code> (string): A JSON formatted string containing information about the server to connect and the node to represent.</li> </ul>	Creates one (or more) new sensor for the device that maps the server (or servers) indicated in the input JSON string. The <code>sensorAlternateId</code> value used is the <code>NodeId</code> of the node on the OPC UA server it maps.
<code>Polling</code>	<ul style="list-style-type: none"> <li>• <code>targetSensor</code> (string): The <code>SensorAlternateId</code> that maps the node on which performs the polling, for example <code>ns=3;i=22</code>.</li> <li>• <code>targetCapability</code> (string): The <code>CapabilityAlternateId</code> that maps the type of the values, for example <code>DoubleValue</code>.</li> </ul>	Creates a new task that performs a read of the value of the node specified by the <code>targetSensor</code> at intervals specified by <code>periodicInterval</code> . As a result, a new measure value is sent for the capability <code>targetCapability</code> of sensor <code>targetSensor</code> with the value read.

Name	Parameters	Description
	<ul style="list-style-type: none"> <li><code>periodicInterval</code> (integer): The interval of time (in milliseconds) between one request for value and the next one.</li> </ul>	
Subscribe	<ul style="list-style-type: none"> <li><code>targetSensor</code> (string): The <code>SensorAlternateId</code> that maps the node on which performs the polling, for example <code>ns=3;i=22</code>.</li> <li><code>targetCapability</code> (string): The <code>CapabilityAlternateId</code> that maps the type of the values, for example <code>DoubleValue</code>.</li> <li><code>publishingInterval</code> (double): Time (in milliseconds) between one publish and the next.</li> <li><code>samplingInterval</code> (double): The interval (in milliseconds) of time between two samples taken.</li> </ul>	Creates a new subscription for the values of the node specified by the <code>targetSensor</code> checking for updates every <code>samplingInterval</code> time. If there is a change, a new measure value is sent for the capability <code>targetCapability</code> of the sensor <code>targetSensor</code> with the value read, every <code>publishingInterval</code> time.
<code>WriteBoolean</code> , <code>WriteDateTime</code> , <code>WriteDouble</code> , <code>WriteInteger</code> , <code>WriteLocalizeText</code> , <code>WriteString</code> , <code>WriteUInteger</code>	<ul style="list-style-type: none"> <li><code>targetSensor</code> (string): The <code>SensorAlternateId</code> that maps the node on which performs the write, for example <code>ns=3;i=22</code>.</li> <li><code>value</code>: The value to write.</li> </ul>	Writes a value of the corresponding type on the node specified by the <code>targetSensor</code> . The node must have write access for the operation to succeed.

## Related Information

[Device Model Mapping \[page 126\]](#)

## 4.10.2 Adapter Configuration

The adapter-specific configuration file providing the information required to connect to each server.

### Secure Connection to the Server

The adapter can establish a secure connection to the OPC UA Server, which authenticates the user. OPC UA applications accept authentication with user name and password, as well as certificate-based authentication (including X.509v3). The digital certificates include a digital signature, which is created by the generator of

the certificate. This digital signature can be self-signed (generated by the private key associated with the certificate) or signed by a certificate authority (generated by the private key associated with the certificate of the certificate authority).

An example of a configuration file is given:

```
{
  "keyStore": {
    "file": "path_to_keystore_file",
    "password": "keystore_password"
  },
  "servers": [
    [
      {
        "protocol": "opc.tcp",
        "ip": "server_ip",
        "port": "server_port",
        "securityPolicy": "Basic256",
        "loginData": {
          "username": "username",
          "password": "password"
        },
        "clientCertificateAlias": "certificate_alias",
        "deviceAlternateId": "my_device",
        "measuresReport": "periodic",
        "periodicInterval": 20000,
        "excludedNSs": [7,8],
        "importAttributes": ["Description", "Historizing"]
      }
    ]
  ]
}
```

The file is in the form of a JSON array and has the following content:

- **keyStore**: this JSON node includes information regarding the keystore the adapter refers to for establishing secure connections to the servers, if needed by the configured security policy. In case only non-secure connections are used, the `keyStore` node can be omitted; but if at least one secure connection needs to be established to a server, then the node must be included and carry the appropriate information. The keystore file must be in a `*.jks` (Java Key Store) format and contain all certificates/private keys needed to communicate to each server. Within a `keyStore` node, the following information must be provided:
  - **file**: a string representing the path to the `keyStore .jks` file. It can be either a relative or absolute path; in the latter case, the path must be specified with respect to the root folder of the Internet of Things Edge Platform deploy (for example, `config/opcuacfg/certificates/clientKeyStore.jks`)
  - **password**: a string representing the password of the `*.jks` keystore file

To use the X.509 IdentityProvider to log on to the OPC-UA server, you need to fill in the `keyStore` JSON node in the configuration file as usual (using values for X.509 certificate).

### Note

Do not insert the `loginData` node. The credentials are pulled from the certificate.

In this way, an X509 IdentityProvider is created based on the credentials contained in the X.509 certificate and used instead of the standard IdentityProvider based on user name and password.

- **servers**: a list of server nodes. Each node has the following content
- **protocol**: the protocol to use for the connection. Shall be set to `opc.tcp`
- **ip**: the server address (or server hostname)
- **port**: the server port

- `securityPolicy`: a string representing the security policy to set up for the connection to the server. The following values can be used:

- `None`
- `Basic128Rsa15`
- `Basic256`
- `Basic256Sha256`

The `securityPolicy` node can be omitted in case communication with the server takes place in an insecure way. This is equivalent to setting `securityPolicy` to `None`

- `loginData`: this JSON node is used to report username and password in case user-based access to the server needs to be performed. It can be omitted, meaning to use anonymous access or X.509 certificate credentials. If specified, the `loginData` node must contain the following elements:
  - `username`: a string representing the user name
  - `password`: a string representing the user password
- `clientCertificateAlias`: a string representing the alias of the private key to be used for securing the communication with this server. A private key associated to this alias must be present in the keystore file referred to by the `keyStore` node. If insecure communication is used to connect to this server, the `clientCertificateAlias` can be omitted.
- `measureReport`: this field regulates the way data retrieval takes place for this server. The possible values for the field are:
  - `periodic`: The Internet of Things Edge Platform polls the server periodically
  - `subscriptionInfo`: The Internet of Things Edge Platform creates a set of monitored items within the server and subscribes to them, to be notified as soon as their value changes
- `periodicInterval`: only relevant in case `measureReport=periodic`. This field regulates the polling interval (in milliseconds) observed by the Internet of Things Edge Platform for the data retrieval
- `subscriptionInfo`: this node is relevant only in case `measureReport=subscription`. It contains additional parameters that regulate the creation of the monitored items the Internet of Things Edge Platform subscribes to. They affect the way the OPC UA server manages the items and the related subscriptions:
  - `samplingInterval`: this parameter regulates the rate (in milliseconds) at which the server checks the data source for changes
  - `publishingInterval`: this parameter regulates the rate (in milliseconds) at which the server sends notifications to the subscribed clients
- `deviceAlternateId`: the physical address to be assigned to the device that maps the server (optional)
- `excludedNSs`: JSON string array containing the namespaces to be excluded from the browse of nodes in the server. To map all nodes regardless of their namespace, set this field to an empty array
- `importAttributes`: JSON string array containing the additional attributes to be managed during the browse of nodes in the server. Such attributes are mapped to custom properties attached to the sensor that maps the OPC UA node. The acceptable values for the array entries map the attribute names defined by the OPC UA specification:
  - `AccessLevel`
  - `ArrayDimensions`
  - `Data Type`
  - `Description`
  - `Historizing`

- `MinimumSamplingInterval`
- `UserAccessLevel`
- `UserWriteMask`
- `ValueRank`
- `WriteMask`

The attached property has the same name of the attribute as specified in the list.

## 4.10.3 Device Model Mapping

To expose the information enclosed in an OPC UA server, the Internet of Things Service performs a normalization of the OPC UA information model that consists in mapping OPC UA specific concepts into the Internet of Things Service concepts. Further information is provided in the following sections.

### 4.10.3.1 Device

The Internet of Things Edge Platform maps an OPC UA server into an Internet of Things Service device. Thus, the resulting OPC UA network will have as many devices as the OPC UA target servers. The device metadata is derived as follows:

- *AlternateId*: it may be retrieved from the adapter-specific configuration file. In case the file does not specify a value, it is set by default as the server URL
- *Protocol address*: it is built as a concatenation of the server hostname/IP and port, in the form `<hostname>/<instance_id>:<port>`
- *Device type*: always set to `Endpoint`
- *Custom properties*: every device has an attached `ServerURL` custom property, which describes the server URL

### 4.10.3.2 Sensor

In general, all OPC UA nodes within a server are mapped to Internet of Things Service sensors, characterized by a set of metadata and properties directly retrieved from the OPC UA node attributes. This behaviour may change depending on the class of the OPC UA node:

- OPC UA nodes of class `ReferenceType` (`class ID 32`) are not mapped to sensors, as they represent relationships between nodes
- OPC UA nodes of class `Variable` (`class ID 2`) are the only one that store a real-world value. As such, they will host a measurement point that represents the real-world value stored in the object

Sensor metadata are derived as follows

- `AlternateId`: set equal to the OPC UA node `nodeId` attribute. The latter is a unique identifier of the OPC UA node within the server and is mandatory for all node classes. It is typically in the form `ns=<namespaceIndex>;<identifiertype>=<identifier>`
- `Name`: set equal to the OPC UA node 'DisplayName' attribute, mandatory for all node classes
- `Custom properties`: they are used to map the values of other OPC UA attributes for the node. In particular:
  - `nodeClass` attribute: mapped to custom property `NodeClass`. It is mandatory for all node classes
  - `BrowseName`: mapped to custom property `BrowseName`. It is mandatory for all node classes.

The attributes `nodeId`, `DisplayName`, `nodeClass`, `BrowseName` are the only attributes that are mandatory for any OPC UA node, regardless of its class. Other attributes may optionally be mapped as well to custom properties, depending on the value of the `importAttributes` entry in the adapter configuration file.

As stated above, sensors corresponding to `Variable` nodes have an attached measurement that refers to a capability defined in the Internet of Things Service data model for the OPC UA protocol. The data model includes a fixed set of measurement capabilities that correspond to the OPC UA data types; their IDs are shown below, together with the Internet of Things Service data type they map to:

- `BooleanValue` (Boolean data type)
- `FloatValue` (Float data type)
- `LongValue` (Long data type)
- `DoubleValue` (Double data type)
- `IntegerValue` (Integer data type)
- `IntegerValue` (unsigned short value)
- `DatetimeValue` (unsigned integer value representing a timestamp)
- `LocalizedTextValue` (UTF-8 string value)
- `StringValue` (UTF-8 string value)

The choice of which capability instance should be attached to the sensor is based on the `DataType` attribute, which is mandatory for `Variable` nodes. In some cases, a certain `Variable` node is writable, that is clients can change its value. In such a case, the sensor will host a command capability instance along with the measurement capability instance. Again, the Internet of Things Service DataModel for the OPC UA protocol lists a set of available commands whose parameters vary depending on the target value data type:

- `WriteBoolean` (Boolean data type)
- `WriteFloat` (Float data type)
- `WriteLong` (Long data type)
- `WriteDouble` (Double data type)
- `WriteInteger` (Integer data type)
- `WriteUInteger` (unsigned short value)
- `WriteDatetime` (unsigned integer value representing a timestamp)
- `WriteLocalizedText` (UTF-8 string value)
- `WriteString` (UTF-8 string value)

## 4.10.4 Mapping Business Objects in the Internet of Things Edge Platform OPC UA

### Background

We define a new approach in mapping OPC UA elements, following the request that only specific tags within the OPC UA server shall be mapped to sensors. In particular, only those tags who represent a real-world or business object. These are typically defined as folder objects in OPC UA structure.

The mapping is implemented by means of a new command exposed by the OPC UA adapter; running this command triggers the creation of as many sensors in the Internet of Things Service as the business objects represented in OPC UA. The command uses a regular expression to identify which OPC UA tags should be mapped, and how to associate them to a `SENSORTYPE` defined by the customer in the Internet of Things Service data model.

### Prerequisites

Before going through the steps mentioned above, users must have onboarded the default OPC UA `SENSORTYPE` as explained in section in the tutorial *Set Up the Internet of Things Edge Platform - OPC UA*.

The default sensor type includes the new command, named `CreateOrUpdateSensors`.

Moreover, since the command maps an OPC UA node to a `SENSOR` able to describe it, a `SENSORTYPE` (together with `CAPABILITIES` and `PROPERTIES`) containing the description of that node must be already present. See above for the details about the structure of the `SENSORTYPE` and how to create it.

### 4.10.4.1 Define New Sensortypes and Capabilities

This section details out the creation of new `SENSORTYPE` and `CAPABILITY` describing a business object. As an example, an "Extruder" machine is considered; the provided payload samples can be further modified to create different types of objects.

The `SENSORTYPE` and `CAPABILITY` creation can take place:

- By means of the Internet of Things Service API, as described in the following subsection "Creation through the API"
- By means of the Internet of Things Service Cockpit: ,

#### Creation Through the API

`SENSORS` has to conform to a `SENSORTYPE` that holds the measurements that the business object is producing. The OPC UA adapter shall be capable of collecting such measurement values from `VARIABLE` tags that are children of the (folder) tag mapped to a `SENSOR`.

An example of such a business object is the `EXTRUDER`. It contains several measurements that are defined as `VARIABLE` type tags in the OPC UA model. To represent this in the new gateway model, a new `SENSORTYPE` representing an `EXTRUDER` needs to be defined.

First, the new capability describing the tags to be monitored needs to be created. Access the Device Management API at <https://<your-instance-id>/<your-tenant>/iot/core/api/v1/doc/swagger>.



The API to create capabilities is:

```
POST /tenant/{tenantId}/capabilities
```

Make sure that the command is invoked with the right `TenantId`, that means, the ID of the tenant where the OPC UA adapter will run. The API payload shall contain the description of the capability to be added.

Example

#### Sample Code

```
{
  "alternateId": "Extruder",
  "name": "Extruder",
  "properties": [
    {
      "name": "Head1",
      "dataType": "float"
    },
    {
      "name": "Head2",
      "dataType": "float"
    },
    {
      "name": "Zone1",
      "dataType": "float"
    },
    {
      "name": "Zone2",
      "dataType": "float"
    },
    {
      "name": "HeadPressure",
      "dataType": "float"
    },
    {
      "name": "VacuumSet",
      "dataType": "float"
    }
  ]
}
```

The following is the definition of a capability with name `Extruder`, extrapolated from the OPC UA Kepware configuration. Each property of the capability describes a measurement produced by the `Extruder`.

Note down the identifier of the created capability, returned in the API response.

The second step to do is to create the `sensorType` itself, adding the created capability specifying the type measure. The API to call is:

```
POST /tenant/{tenantId}/sensorTypes
```

Again, make sure that the command is invoked with the right `TenantId`. In the following API payload example, the same name "Extruder" given to the capability is used also for the `SensorType`, but this is not a requirement, there is freedom of choice for names. The "id" field must include the identifier of the capability created through the `POST /capabilities` API. The `alternateId` field in the payload can be omitted; in such a case, the `alternateId` for the `SensorType` is set by the system.

#### Sample Code

```
{
  "alternateId": "6333",
```

```

    "name": "Extruder",
    "capabilities": [
      {
        "id": "81d34073-e5cb-4547-b48c-accba19f8e5e",
        "type": "measure"
      }
    ]
  }
}

```

## 4.10.4.2 New Command

The following is the definition of the capability corresponding to the command, with all the parameters:

### Sample Code

```

{
  "alternateId": "CreateOrUpdateSensors",
  "name": "CreateOrUpdateSensors",
  "properties": [
    {
      "name": "rootNodeId",
      "dataType": "string"
    },
    {
      "name": "sensorRegex",
      "dataType": "string"
    },
    {
      "name": "sensorTypeAlternateId",
      "dataType": "string"
    },
    {
      "name": "capabilityAlternateId",
      "dataType": "string"
    },
    {
      "name": "propertyRegex",
      "dataType": "string"
    },
    {
      "name": "periodicInterval",
      "dataType": "integer"
    },
    {
      "name": "publishingInterval",
      "dataType": "double"
    },
    {
      "name": "samplingInterval",
      "dataType": "double"
    }
  ]
}

```

The command `CreateOrUpdateSensors` contains both the parameters required to configure a periodic reporting of measures (`periodicInterval`) and for subscription (`publishingInterval`, `samplingInterval`). The use of the command contemplates to use just one of the options, not both together. The following is the full list of parameters:

- **rootNodeId**: The `nodeId` of the node used to start searching for the tags to be onboarded. This is useful to limit the search space inside the OPC UA server structure, if known a priori. To start from the root node, containing the whole object space, the `nodeId` of root node ("**ns=0;i=85**") can be used. In case the `nodeId` of a different node is provided, the search will start from that node's children nodes.
- **sensorRegex**: A regular expression used to select the nodes that will be mapped as `Sensor`. The regular expression is applied to the node's `browseName` attribute; in case it matches, the node will be mapped as a `Sensor`.
- **sensorTypeAlternateId**: The `alternateId` value of an existing `SensorType`. The node will be then onboarded as a `Sensor` conforming to that `SensorType`.
- **capabilityAlternateId**: The `alternateId` value of a capability defined under the `SensorType` with `sensorTypeAlternateId` specified in the command.
- **propertyRegex**: A regular expression used to determine the property name of the measurement produced by a certain node. Once an OPC UA node is eligible for the onboarding as a `Sensor`, a search through its children nodes is performed to determine which of them is producing measurements. A node will be used for ingestion if it matches the following criteria:
  - It is an OPC UA Variable Node
  - Applying the "propertyRegex" on the node `browseName` returns the name of a valid property, defined in the selected capability of the target `SensorType`
 In case both conditions apply, a subscription is created on the node, which allows the Internet of Things Edge Platform to receive the node value as soon as it changes over time. The value received from the subscription is associated to the property whose name is retrieved by applying the "propertyRegex" on the node `browseName`.
- **publishingInterval** and **samplingInterval** values used to create the subscription in the OPC UA server. If they are specified, the reporting mechanism used is subscription.
- **periodicInterval**: The time between the sending of a measure and the next one. If it is specified, the reporting mechanism used is polling.

### Example

Continuing the example of the `Extruder` object, suppose that the OPC UA server connected contains two objects `Extruder`, with names `Extruder1` and `Extruder2`, and that they definition fits with the `SensorType` `Extruder` (with `alternateId` value 18) defined above.

An example of the syntax for the command `CreateOrUpdateSensors`, would be the following:

#### Sample Code

```
{
  "capabilityId": "aa8342d5-c57d-4a21-9998-506c2a810768",
  "command": {
    "rootNodeId": "ns=0;i=85",
    "sensorRegex": "Extruder.",
    "sensorTypeAlternateId": "18",
    "capabilityAlternateId": "Extruder",
    "propertyRegex": "Head.*",
    "publishingInterval": 100.0,
    "samplingInterval": 1000.0
  },
  "sensorId": "770d7b9a-c58e-4a8a-b9a0-5c7ea7b8c153"
}
```

The result of the invocation of this command will be the creation of two `Sensors` in the gateway environment, corresponding to the two objects `Extruder1` and `Extruder2` defined in the OPC UA server. Moreover, each of

the `Sensors` have a subscription for properties `Head1` and `Head2`, with the configuration timing defined. For more information on the `Sensor` objects, please refer to section [Sensors \[page 132\]](#).

### 4.10.4.3 Sensors

The sensors created by the command `CreateOrUpdateSensors` contains several custom properties used to store information about the OPC UA node it represents, and to implement the reporting technique defined for it (subscription or polling).

In details, the custom properties that are always present on a `Sensor`, and are related to the OPC UA object are:

- `BrowseName`: The one defined in the OPC UA server.
  - Example: `Extruder1`
- `NodeId`: The `nodeId` defined in the OPC UA server.
  - Example: `ns=2;s=Extruder1`
- `ParentNodeId`: The `NodeId` of the parent node, as defined in the OPC UA server.
  - Example: `ns=0;i=85`

#### SubscriptionInfo

In case of subscription, a custom property for each monitored item in the OPC UA server in subscription mode. In details, the key of the property will be in the form `SubscriptionInfo-<NodeId>` of the tag monitored, and the value will be a JSON-formatted string containing the details of the subscription and also of the capability and property in the `SensorType`.

The following is an example of a custom property for the node `Head1` under the `Extruder` object.

#### Sample Code

```
Key:
  SubscriptionInfo-ns=2;s=Extruder1.Brand.Head1
Value:
  {
    "subscriptionId":122,
    "publishingInterval":100.0,
    "samplingInterval":1000.0,
    "capabilityAlternateId":"Extruder",
    "propertyName":"Head1"
  }
```

#### PollingInfo

In case of polling, a custom property for each monitored item in the OPC UA server in subscription mode. In details, the key of the property will be in the form `PollingInfo-<property in the SensorType>`, and the value will be a JSON-formatted string containing the `NodeId`, the capability in the `SensorType`, and `periodicInterval` value.

The following is an example of a custom property for the node `Head1` under the `Extruder` object.

#### Sample Code

```
Key:
```

```
    PollingInfo-Head1
  Value:
    {
      "nodeId":ns=2;s=Extruder1.Brand.Head1,
      "capabilityAlternateId":"Extruder",
      "periodicInterval":5000.0
    }
```

## 4.11 SigFox

### Note

The SigFox adapter is only available for the Internet of Things Edge Platform. For more information, please refer to the tutorial [Set Up the Internet of Things Edge Platform - SigFox](#).

## Related Information

[Overview \[page 133\]](#)

[Architecture \[page 134\]](#)

[Adapter Configuration \[page 135\]](#)

[Security \[page 140\]](#)

[Device Features \[page 141\]](#)

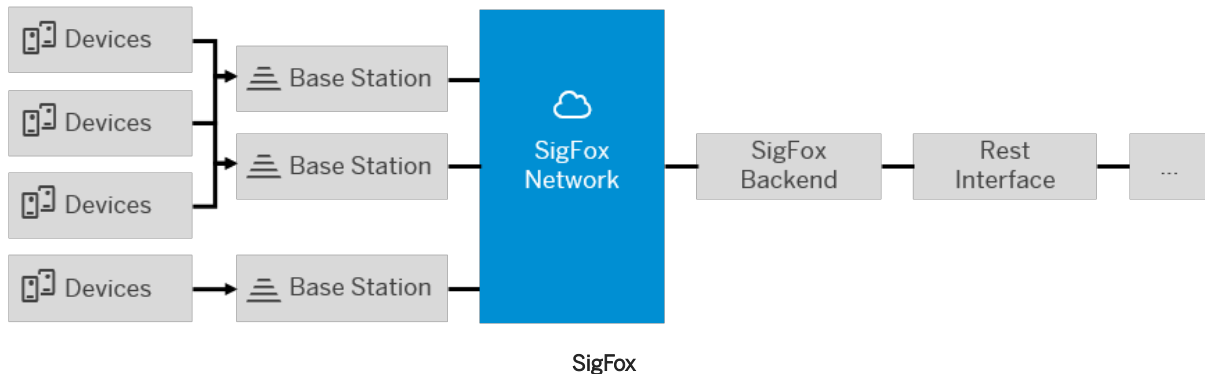
[APIs \[page 142\]](#)

[Cloud Entities \[page 142\]](#)

[Device Registration \[page 142\]](#)

### 4.11.1 Overview

SigFox is a French company employing a UNB-based (Ultra Narrow Band) radio technology to connect IoT devices to its network. It currently uses the ISM band on 868MHz (as defined by ETSI and CEPT) in Europe and on 902MHz in the US (as defined by the FCC), depending on specific regional regulations. To expand the service coverage area, SigFox has partnerships with several network operators around the world. By utilizing the UNB infrastructure provided by these partners, SigFox enables communication between devices operating the SigFox radio protocol and the SigFox cloud. In this context, the SigFox infrastructure acts as the transport layer which sends the data generated by the devices to the cloud. This layer is data-agnostic, meaning that it does not interpret the data but simply transfers it. Considering this, device manufacturers who install the SigFox firmware on their products can use the SigFox service to access their data, and a set of REST APIs to build applications based on this data. The graphic illustrates the SigFox network in a simplified way.



The SigFox-enabled devices connect to a local base station using the proprietary SigFox radio protocol. The data produced by each device is pushed to the SigFox backend, which is the only interface available for integrators to onboard, manage, and control the devices. The interfacing relies on the set of REST APIs published by the backend server. This approach abstracts network and connectivity aspects to such a degree that device users can focus on their specific business logic. Network metrics like SNR and base station IDs are still available to provide a minimum set of information for deployment and troubleshooting purposes.

## 4.11.2 Architecture

The SigFox adapter allows the Internet of Things Gateway Edge to interface with the SigFox cloud service. Since the communication with SigFox relies on HTTP/REST calls, the SigFox adapter is built on top of common REST functionalities and is a special variant of the REST protocol.

### 4.11.2.1 Interaction with the SigFox Infrastructure

Interaction with the SigFox environment can take place through two different channels:

- **Using REST APIs:** the client implements a REST endpoint which connects to the SigFox REST server. HTTPS with HTTP Basic authentication is required.
- **Using callback APIs:** this pattern is mainly used for data retrieval. The user implements a REST callback server while the SigFox infrastructure operates a REST client. This client calls the user-defined APIs published by the callback server, with the relevant data as API content. In this scenario, security mechanisms (HTTPS-secured channel) are optional. Callback APIs are also used to push data to the devices.

Data can be retrieved both through polling and callback.

However, so far, the Internet of Things Gateway Edge supports only the polling methodology, behaving like a REST client with an 'uplink' communication towards the SigFox cloud (server). As a client, the Internet of Things Gateway Edge consumes SigFox APIs to retrieve topology information on the visible device types and devices, to send commands, and to retrieve device data in the polling mode.

## 4.11.2.2 Management of SigFox Messages

As described in the previous section, the SigFox infrastructure does not define a strict message payload format which may be freely used by device manufacturers to send data to the backend. Therefore, a flexible and extensible framework for managing the message payload is required. The configuration file in the Internet of Things Gateway Edge makes it possible to split up 12-byte messages into a configurable number of sub-arrays of different size. Each message part is then sent to the DataModel bundle (running on the Internet of Things Core Service by default) where it is parsed, provided that any meaningful data can be derived. This policy is applied to all message which are structured as a series of self-contained fields. Each message is either split on a position basis (if it is composed of a series of fixed-length fields) or by using a special character sequence as a token (an example are textual protocols where fields are separated by a specific character). The Internet of Things Gateway Edge automatically assigns capability instances to the measurements that result from splitting the incoming payload.

## 4.11.3 Adapter Configuration

Being a specialization of the REST protocol adapter, the SigFox adapter uses the common configuration file for REST. This file is called `config_gateway_rest.xml` and can be found under `<gateway_deploy>/config`. The file comprises only generic configuration data. There is also an additional configuration file for SigFox-specific metadata.

### 4.11.3.1 Generic REST Configuration File

This is the most important part of the `config_gateway_rest.xml` file within a SigFox deployment:

```
<cnf:gateway>
  <cnf:technology>rest</cnf:technology>
  <cnf:vendor>sigfox</cnf:vendor>
  <cnf:gateway_comm>
    <cnf:media>API</cnf:media>
    <cnf:useAsServer>true</cnf:useAsServer>
    <cnf:socket_host>https://backend-demo.sigfox.com</cnf:socket_host>
    <cnf:socket_port>80</cnf:socket_port>
    <cnf:apiUsername>insert_your_username</cnf:apiUsername>
    <cnf:apiPassword>insert_your_password</cnf:apiPassword>
    <cnf:disableGatewayContentSent>true</cnf:disableGatewayContentSent>
  </cnf:gateway_comm>
  ...
</cnf:gateway>
```

This section must be filled with data related to the SigFox account which is used to consume the SigFox service. Typically, this information can be retrieved either directly from SigFox or from a customer who has a valid SigFox account. The entries have the following functions:

- `cnf:technology`: must be set to `rest`. Identifies the underlying technology (a RESTful web service).
- `cnf:vendor`: must be set to `sigfox`. Identifies the vendor for the specific REST service.
- `cnf:API`: must be set to `API`. All interaction with the service is based on APIs.

- `cnf:useAsServer`: must be set to `true`
- `cnf:socket_host`: must contain the URL of the remote service. In the case of SigFox, it is usually mapped to either the demo backend (<https://backend-demo.sigfox.com/>) or the production backend (<https://backend.sigfox.com/>).
- `cnf:socket_port`: contains the port for the remote service. Must be set to 80 for the SigFox backend.
- `cnf:apiUsername`: the user name used for authentication at the SigFox backend
- `cnf:apiPassword`: the password used for authentication at the SigFox backend

The values under `cnf:apiUsername/cnf:apiPassword` are used to fill the Authorization header of every API call sent to the SigFox backend; `username` and `password` will be Base64-encoded according to the Basic Authentication strategy as required by SigFox.

Below is another part of the `config_gateway_rest.xml` file which is specific for the SigFox adapter:

```
<cnf:gateway>
  ...
  <cnf:defaultValues>
    <cnf:defaultSensorTypeAlternateId>1</cnf:defaultSensorTypeAlternateId>
  </cnf:defaultValues>
</cnf:gateway>
```

The default `sensor type alternate ID` for Sigfox is identified by the value 1: This ensures that the default value 0 will continue to be used to identify the default profile for the generic REST protocol.

### 4.11.3.2 SigFox-Specific Configuration File

The SigFox adapter requires additional configuration data to be specified along with the usual Internet of Things Edge Platform configuration. This data is retrieved from the JSON-formatted `sigfox.json` configuration file stored in the `<gateway_deploy_folder>/config/restConfig` folder.

A configuration template for the `sigfox.json` file is given below:

```
{
  "discoveryDevicesPollingInterval": null,
  "deviceTypes" :
  [{
    "id" : "56f16036e4b02027a92459cf",
    "iotServiceDeviceType": "WaterMeter",
    "pollingApiInterval": 60000,
    "sensorTypeAlternateId" : "4",
    "stringMeasure" : false,
    "msbFirst": false,
    "frameSelectionByteStart": 0,
    "frameSelectionBitStart": 7,
    "frameSelectionBitSize": 8,
    "frameSelectionParsing": "Integer",
    "frameParsingData": [{
      "frameSelectionValue": "2",
      "sensorAlternateId": [ "1", "1", "1", "1", "1", "1", "1", "1" ],
      "arrayChunkByteStart" : [ 1, 1, 1, 1, 1, 2, 3 ],
      "arrayChunkBitStart": [ 0, 1, 2, 3, 4, 7, 7 ],
      "arrayChunkBitSize" : [ 1, 1, 1, 1, 4, 8, 32 ],
      "measureByteSize": [ 1, 1, 1, 1, 1, 1, 8 ],
      "measureCapabilityAlternateId" : [ "22", "23", "24", "25", "26",
"6", "8" ]
    }],
  }, {
```



```

"frameSelectionValue": "3",
"sensorAlternateId": [ "1", "1", "1", "1", "1", "1", "1", "1", "1" ],
"arrayChunkByteStart": [ 1, 1, 1, 1, 1, 2, 3, 5, 6 ],
"arrayChunkBitStart": [ 0, 1, 2, 3, 4, 7, 7, 7, 7 ],
"arrayChunkBitSize": [ 1, 1, 1, 1, 4, 8, 16, 8, 8 ],
"measureByteSize": [ 1, 1, 1, 1, 1, 1, 4, 1, 1 ],
"measureCapabilityAlternateId": [ "22", "23", "24", "25", "26",
"10", "11", "12", "13" ]
}, {
"frameSelectionValue": "4",
"sensorAlternateId": [ "1", "1", "1", "1", "1", "1" ],
"arrayChunkByteStart": [ 1, 1, 1, 1, 1, 2 ],
"arrayChunkBitStart": [ 0, 1, 2, 3, 4, 7 ],
"arrayChunkBitSize": [ 1, 1, 1, 1, 4, 16 ],
"measureByteSize": [ 1, 1, 1, 1, 1, 4 ],
"measureCapabilityAlternateId": [ "22", "23", "24", "25", "26",
"15" ]
}],
"commandSensorAlternateId": [ "0" ],
"commandSensorTypeAlternateId": [ "1" ],
"commandCapabilityAlternateId": [ "20" ]
}
}

```

### 4.11.3.2.1 Node Discovery With Timer Task

You can use the optional parameter

- `discoveryDevicesPollingInterval`

to enable the discovery of nodes using a timer task which runs at the `discoveryDevicesPollingInterval` interval specified [in milliseconds].

### 4.11.3.2.2 Data Polling Task

If you want to configure the adapter to retrieve data through polling (see the template of the `sigfox.json` configuration file), you must specify the recurrence of the polling task.

You can do this using the parameter

- `pollingApiInterval`

This parameter allows you to enable the data polling task on a device basis (which means the task is always associated with the default sensor for the given device), running at the `pollingApiInterval` interval you define [in milliseconds].

You can enable the data polling task for all devices of a specific device type. Once polling has been activated in the configuration file, you can change the polling interval in the Internet of Things Service Cockpit, if required. To do this, adjust the `Report Cycle` of the device (by selecting the default sensor of the device).

### 4.11.3.2.3 Measurement Handling Configuration

The Sigfox-specific configuration file contains a section which controls how messages are parsed and mapped to specific sensors/capabilities. In particular, these entries allow you to identify a specific set of data within a SigFox message. This set of data is then parsed and associated with a specific data capability (identified by a `sensorTypeAlternateId/capabilityAlternateId` couple) within a sensor. A capability instance is added to the device/sensor instance when the first data is received. The following fields control measurement handling:

- `sensorTypeAlternateId`: the sensor type alternate id with which this data callback is associated
- `stringMeasure`: `true` if the data must be handled as a string, otherwise `false` (for example, if the data must be handled as a byte array)

### 4.11.3.2.4 Device Payload Handling

SigFox devices output data in chunks of 12 bytes. The SigFox adapter handles string representations of such byte arrays and can be instructed how to parse them on a device type basis. The configuration file part, which controls this is shown below:

```
"frameSelectionByteStart": 0,
  "frameSelectionBitStart": 7,
  "frameSelectionBitSize": 8,
  "frameSelectionParsing": "Integer",
  "frameParsingData": [{
    "frameSelectionValue": "2",
    "sensorAlternateId": [ "1", "1", "1", "1", "1", "1", "1", "1" ],
    "arrayChunkByteStart": [ 1, 1, 1, 1, 1, 2, 3 ],
    "arrayChunkBitStart": [ 0, 1, 2, 3, 4, 7, 7 ],
    "arrayChunkBitSize": [ 1, 1, 1, 1, 4, 8, 32 ],
    "measureByteSize": [ 1, 1, 1, 1, 1, 1, 8 ],
    "measureCapabilityAlternateId": [ "22", "23", "24", "25", "26",
"6", "8" ]
  }],{
    "frameSelectionValue": "3",
    "sensorAlternateId": [ "1", "1", "1", "1", "1", "1", "1", "1", "1", "1" ],
    "arrayChunkByteStart": [ 1, 1, 1, 1, 1, 2, 3, 5, 6 ],
    "arrayChunkBitStart": [ 0, 1, 2, 3, 4, 7, 7, 7, 7 ],
    "arrayChunkBitSize": [ 1, 1, 1, 1, 4, 8, 16, 8, 8 ],
    "measureByteSize": [ 1, 1, 1, 1, 1, 1, 4, 1, 1 ],
    "measureCapabilityAlternateId": [ "22", "23", "24", "25", "26",
"10", "11",
"12", "13" ]
  }],{
    "frameSelectionValue": "4",
    "sensorAlternateId": [ "1", "1", "1", "1", "1", "1" ],
    "arrayChunkByteStart": [ 1, 1, 1, 1, 1, 2 ],
    "arrayChunkBitStart": [ 0, 1, 2, 3, 4, 7 ],
    "arrayChunkBitSize": [ 1, 1, 1, 1, 4, 16 ],
    "measureByteSize": [ 1, 1, 1, 1, 1, 4 ],
    "measureCapabilityAlternateId": [ "22", "23", "24", "25", "26",
"15" ]
  }],
}
```

The following section is used to retrieve the frame type indicator:

- `frameSelectionByteStart`: indicates the start byte for the frame type indicator

- `frameSelectionBitStart`: indicates the start bit within the byte specified in `frameSelectionByteStart` for the frame type indicator. A big endian format for the byte representation is expected.
- `frameSelectionBitSize`: indicates the number of bits which make up the frame type indicator
- `frameSelectionParsing`: indicates how to parse the bits for the frame type indicator

The example above fits a use case where the data sent by the specific device type always has the frame type indicator in the first byte of the raw data string. Byte 0 is addressed with all its bits (the start bit is 7, the first in a big endian sequence, and the bit length is 8). The extracted byte is then parsed as an integer (string parsing is also allowed) to retrieve the frame type indicator.

Once a frame type indicator is available, its value is used to select a node from the list of `frameParsingData` items, based on the value of the `frameSelectionValue` node in each `frameParsingData` node. Basically, a `frameParsingData` node is designed to represent a specific frame structure and provide a way to handle this structure, in short, a way to extract the byte chunks for each single measurement from the raw data byte array. The fields in a `frameParsingData` node have the following function:

- `frameSelectionValue`: contains the frame type indicator value associated with this frame parsing data
- `sensorAlternateId`: contains an array of string values representing the sensor alternate ids with which the measurements will be associated
- `arrayChunkByteStart`: contains an array whose items represent the start byte for each measurement
- `arrayChunkBitStart`: contains an array whose items represent the start bit within the corresponding `arrayChunkByteStart` for each measurement
- `arrayChunkBitSize`: contains an array whose items represent the bit size within the raw data for each measurement
- `measureByteSize`: contains an array whose items represent the size of the byte array produced for each measurement. The bits extracted according to the instructions in the `arrayChunk_XXX` fields will be copied into a byte array of size `measureByteSize` and then wrapped in a `measure`.
- `measureCapabilityAlternateId`: contains an array of string values whose items represent the alternate capability ID for the measurement produced by the byte chunk

In general, if the specific frame structure comprises N different measurements, these will be backed by N byte array chunks. This way, the arrays within a `frameParsingData` node must include N items each. If the array rank is different across the nodes of a specific `frameParsingData` node, a configuration file consistency error will occur, preventing the Internet of Things Edge Platform from starting correctly. Referring to the example above, the measurements will be generated according to 3 different frame types, denoted as 2, 3 and 4. Frame type 2 produces 7 measurements. All of them are associated with the sensor having the alternate id 1 but they all have a different measure alternate capability. The first 5 of them will be extracted from byte 0: 4 are just single-bit flags at different positions within byte 0 (their bit size is 1), one is instead taken from the second nibble in byte 0 (bit size is 4, bit start is 4).

Some device applications use a constant framing format so that there is no need to identify a specific frame type indicator in the raw data. If this is the case, the 4 `frameSelection_XXXX` fields must be omitted, and the `frameParsingData` array node must contain a single item only.

### 4.11.3.2.5 Management of Commands

Use this part of the device type-level configuration to add a set of generic commands to the devices belonging to this device type. The result is a command capability instance which is attached to a specific sensor for each

device of this type. It is possible to add commands defined in a dedicated, customer-specific profile. In such a case, a configuration node in the file instructs the Internet of Things Edge Platform how to handle command arguments. Below please find the section of the configuration file which controls command management:

```
"commandSensorAlternateId" : [ "0", "2" ],
  "commandSensorTypeAlternateId": [ "1", "4" ],
  "commandCapabilityAlternateId" : [ "20", "70" ],
  "commandParsingData": [{
    "commandCapabilityAlternateId" : "70",
    "arraySize" : 8,
    "arrayChunkByteStart" : [ 0, 1, 2, 4, 5, 6 ],
    "arrayChunkBitStart": [ 7, 7, 7, 7, 7, 7 ],
    "arrayChunkBitSize" : [ 8, 8, 16, 8, 8, 8 ],
    "valueTypes" : [ "INT8", "genCmdParam", "genCmdParam",
"genCmdParam", "genCmdParam", "genCmdParam" ],
    "values" : [ "3", "AMRType", "transmitPeriod", "channelOnOff",
"channel1Type", "channel2Type" ]
  }],
```

- `commandSensorAlternateId`: an array of string values, each representing the identifier of the sensor where the command will be added
- `commandSensorTypeAlternateId`: an array of string values, each representing the `sensor type alternate id` for the generic command to be added
- `commandCapabilityAlternateId`: an array of string values, each representing the `capabilityId` of a generic command
- `commandParsingData`: an array describing the way how to parse the command associated with the corresponding `commandCapabilityAlternateId` (this section is optional and is required only if you want to manage specific commands to be sent to the device) [not valid for simulated devices]

## 4.11.4 Security

The SigFox service addresses security both at API level (link from client to SigFox backend) and radio level (link from device to the SigFox backend). The latter is not relevant in the context of the SigFox adapter, which has no direct connection with devices.

### 4.11.4.1 Radio Level Security

The SigFox protocol uses a simple but efficient state-of-the-art HMAC payload check with a private key for each device. It ensures both authentication and payload integrity (along with CRC), and also prevents message forging and replay.

### 4.11.4.2 API Level Security

The SigFox service is accessible only through HTTPS, and all API endpoints require authentication credentials (login with password). Authentication is performed using the simple HTTP Basic scheme. The authentication credentials are assigned to a group which determines the device types available with these credentials.

### 4.11.4.3 Device Onboarding

The following steps are required to onboard new devices on the SigFox network:

1. Purchase SigFox-enabled hardware.
2. Obtain the addressing information of the device: identifier (device ID) and PAC code for each device.
3. Sign a subscription contract with SigFox.
4. Request SigFox credentials for access to the backend.
5. Register the devices using their ID and PAC, and assign them to a device type.

### 4.11.5 Device Features

Devices implementing the SigFox radio technology are used in a wide range of applications, including smart cities, smart agriculture, asset tracking and similar. The features of the UNB communications allow to keep resource consumption on the device at a minimum level, allowing for a long battery lifecycle. The following sections provide additional details of some device features of interest.

#### 4.11.5.1 Communication Paradigms

Like other competing UNB approaches (for example, LoRa), the SigFox solutions focus on low throughput. In the uplink direction (FROM device TO network), a SigFox device can send between 0 and 140 messages a day, with each message carrying up to 12 bytes of actual payload data. In the downlink direction (FROM network TO device), the SigFox network structure ensures that up to 4 messages with 8 bytes of payload are transmitted to each device every day. Any additional downlink message will be handled by the network in a best-effort fashion. In general, any downlink communication must be initiated by the device: to receive messages, the device must first send data through the uplink channel and then request downlink data from the network. In other words, it is not possible to send data or commands to a device at all times, because the device first needs to open an uplink channel. This is done whenever the device has data to dispatch, or if it is configured to receive a downlink message at specific events or times.

As said above, the SigFox network is payload-agnostic because it does neither parse nor transform a message payload: the protocol-specific structure of a message is only disclosed to the device manufacturer and its partners or customers. A manufacturer is free to decide how to use the 12 bytes of data contained in the payload.

#### 4.11.5.2 Power Consumption

The communication paradigms described above are designed to minimize power consumption to increase the battery lifetime. Moreover, SigFox-enabled modems operate with low transmission power and without

continuous network synchronization. This increases energy efficiency considerably and ensures long-term autonomy (the battery lifetime is obviously affected by the size and frequency of messages, as well as the antenna emission power).

### 4.11.5.3 Device Classes

SigFox defines four device classes (0, 1, 2, 3) based on device radio performance values. Class 0 provide the highest radio quality, and class 3 the lowest. A class 3 device is generally perfectly suitable for outdoor deployment in well-covered regions, but will not necessarily operate correctly indoors. Consequently, since a class 0 device shows the best radio capability, it is better suited for more complex deploy scenarios from a radio communication point of view, such as deep indoor locations.

### 4.11.6 APIs

SigFox REST APIs can be grouped into the following categories:

- Group APIs: used to create and manage **groups** which represent sets of SigFox users with an authorized account on the SigFox backend
- Device types APIs: used to create and manage device **types** which represent sets of devices with similar features according to specific user' criteria
- Devices APIs: used to manage single devices, usually identified by their unique ID

### 4.11.7 Cloud Entities

The SigFox cloud defines a series of entities which represent both the actors involved in device management and the categories used to group devices. The relevant entities for the SigFox plug-in are:

- **device**: represents a network device which is mapped to an Internet of Things Service device. It is uniquely identified by a device ID mapped to the Device Alternate Id of the Internet of Things Service device.
- **device type**: represents a category of devices, for example devices with a specific purpose and identical features (such as temperature sensors from vendor X)
- **group**: a SigFox customer who has access to the SigFox backend and has visibility rights for specific devices and device types

### 4.11.8 Device Registration

This section briefly describes the process of onboarding a SigFox device so that its data can be retrieved by the Internet of Things Service. The actors involved are:

- **The device manufacturer / reseller.** They provide:
  - SIGFOX-enabled object hardware
  - Object device ID and PAC code
- **The SIFOX commercial team.** It provides:
  - SIGFOX subscription
  - SIGFOX login
- **The device user.** They are responsible for registering their devices.

## 4.11.8.1 Object Registration

This phase allows you to add a new device to the set of operational devices for a user so that it can be successfully addressed and inspected by means of the SigFox APIs and the Internet of Things Service. Two basic operations are needed.

### 4.11.8.1.1 Create a Device Type

This command creates a new device type, that is, a set of devices with similar features. This step is needed if a new device cannot be assigned to any existing device type in the SigFox account of the user.

### 4.11.8.1.2 Register a New Device

This API allows you to actually onboard a new device and assign it to a specific device type. You need an ID-PAC pair to register the device successfully.

## 4.12 SNMP

The specialization of Internet of Things Edge Platform for SNMP adapter allows you to gather information from SNMP-enabled devices. In this scenario, the Internet of Things Edge Platform acts as an SNMP manager that retrieves data from the SNMP agents running on the network devices.

### Note

The SNMP adapter is only available for the Internet of Things Edge Platform. For more information, please refer to the tutorial [Set Up the Internet of Things Edge Platform - SNMP](#).

## Related Information

[Overview \[page 144\]](#)

[Adapter Configuration \[page 144\]](#)

[Network Addressing \[page 145\]](#)

[Node Discovery \[page 146\]](#)

### 4.12.1 Overview

The data flow follows two routes:

- The Internet of Things Edge Platform issues a cyclic polling over all the devices, performing a complete MIB-WALK for each connected device.
- The Internet of Things Edge Platform receives traps from devices, if the devices are able to send the traps.

#### Note

Only communication from the SNMP client to the Internet of Things Edge Platform is supported. Communication from the Internet of Things Edge Platform to the SNMP client, in form of generic commands for example, is not supported.

### 4.12.2 Adapter Configuration

To configure the Internet of Things Gateway SNMP adapter, edit the relevant configuration file.

The XML file has to contain a root element `<cnf:configuration>` with all the necessary configurations.

1. Specify the necessary parameters to establish a connection. For SNMPv2c, set the parameters like in the following example.

```
<cnf:snmpIPAddr>127.0.0.1</cnf:snmpIPAddr>
  <cnf:snmpPort>161</cnf:snmpPort>
  <cnf:snmpPortTrap>162</cnf:snmpPortTrap>
  <cnf:pollingTime>60000</cnf:pollingTime>
  <cnf:snmpVersion>2c</cnf:snmpVersion>
  <cnf:community>public</cnf:community>
  <cnf:securityName>private</cnf:securityName>
```

2. Specify the device alternate id using the mandatory element `<cnf:alternateId>`. You can specify an OID to get the address directly from the MIB, or a specific address. To do so, use the `oid` or the `defined` attributes. If both attributes are specified, `oid` has the priority.

The following are two different examples showing the use of the attributes `oid` and `defined`:

```
<cnf:alternateId oid=".1.3.6.1.2.1.1.4.0" />
  <cnf:alternateId defined="myaddress" />
```



3. Specify the timestamp and the value for each measure, giving a reference in the MIB. The following is the syntax for the timestamp, where the attribute type is specified with the value `UnixTimestamp`. An `<cnf:rowID>` element is inserted for each measure.

```
<cnf:measureTimestamp oid=".1.3.6.1.2.1.1.4" type="UnixTimestamp" >
  <cnf:rowID>0</cnf:rowID>
</cnf:measureTimestamp>
```

4. Specify the measures with their identifier (`capabilityId`) and the `dataType`.

```
<cnf:measure oid=".1.3.6.1.2.1.1.4" capabilityAlternateId="12"
dataType="OctetString">
  <cnf:rowID>0</cnf:rowID>
</cnf:measure>
```

## 4.12.3 Network Addressing

Within the , devices managed by the Internet of Things Gateway SNMP can be addressed using the following values.

- `AlternateId`: the identifier of the device running the SNMP agent.
- `ProtocolAddress`: represents the IP address of the device running the SNMP agent and TCP port of the device running SNMP agent.
- `SensorAlternateId`: the object identifier (OID) for the current leaf of the management information base (MIB) visited by Internet of Things Gateway.
- `CapabilityAlternateId`: the type of data published in the current leaf of the MIB (that is, `Integer32`, `OctetString`, and others).  
Every sensor has only `capabilityAlternateId`. Moreover, the name and description of the sensors are retrieved from the MIB of the device.

### 4.12.3.1 Object ID Mapping

The following values of the object ID are supported:

- 2: integer 32 bits (MIB: `Integer32`)
- 4: string (MIB: `OCTET STRING`)
- 6: MIB Object Identifier (MIB: `OBJECT IDENTIFIER`)
- 64: IP address (MIB: `IPADDRESS`)
- 65: integer counter 32 bits, same as integer (MIB: `COUNTER32`)
- 66: unsigned integer 32 bits (MIB: `UNSIGNED INTEGER32`)
- 67: timestamp (MIB: `TIMETICKS`)
- 70: integer counter 64 bits, same as integer (MIB: `COUNTER64`)
- 130: boolean (MIB: `BOOLEAN`)
- 131: `TruthValue`, same as boolean (MIB: `TruthValue`)

## 4.12.4 Node Discovery

To discover new nodes, you manually send a generic command. This generic command has to contain the following information:

- IP / TCP port of the device under discovery
- MIB that the device under discovery should publish
- list of OIDs to be checked, together with the related ObjectID that allows linking the current OID to the WSNDatamodel parser instrumentation
- SNMP version of the device under discovery
- credentials to be enabled to read from the device
- (optional) polling time to retrieve info from the device; if not provided, default is 60 seconds

The file containing this information is stored in the `config/snmpDevices` folder of the Internet of Things Gateway instance and is used in the following start-ups of this instance. Note that there is only one file for each managed device. The name of the file follows the pattern `snmpConfig-<ipAddr>-<tcpPort>.xml`, where `<ipAddr>` is the physical address of the device (in decimal form and separated by '\_'), and `<tcpPort>` is the TCP port of the machine running the SNMP agent of the device (in decimal form).

# 5 Internet of Things Edge Platform Scalability

The Internet of Things Edge Platform can be configured to run in a scale-out mode where multiple instances are federated in a cluster to share the processing load.

This documentation focuses on the Internet of Things Edge Platform MQTT and REST adapters, as they implement the most popular protocols to connect a device to an Internet of Things Service instance.

The first part of this documentation provides an overview on the basic concepts and architecture of the Internet of Things Edge Platform scale-out deploy. The second part of this documentation contains configuration guidelines to implement an Internet of Things Edge Platform scale-out deploy. For simplicity, the documentation details a setup with two Internet of Things Edge Platform instances in the cluster; nevertheless, no functional limitations exist that prevent from adding new instances to the cluster.

## Related Information

[Architectural Overview \[page 147\]](#)

[Internet of Things Edge Platform Configuration \[page 150\]](#)

[MQTT-Specific Configuration \[page 154\]](#)

[REST-Specific Configuration \[page 157\]](#)

[Deployment of Custom Bundles \[page 157\]](#)

[Internet of Things Edge Platform Upgrade \[page 158\]](#)

[Internet of Things Edge Platform Startup \[page 159\]](#)

[Networking Setup \[page 160\]](#)

## 5.1 Architectural Overview

### 5.1.1 Limitations

The scale-out setup introduced in this section represents a solution to share the processing load for MQTT and REST device measurement ingestion and command delivery across a set of clustered Internet of Things Edge Platform instances. As such, it has the following limitations:

- It doesn't cover other ingestion protocols.

- It doesn't support the management of custom bundles using the set of APIs under `/tenant/{tenantId}/gateways/{gatewayId}/bundles`. Custom bundles can still be installed manually as explained in the following documentation.

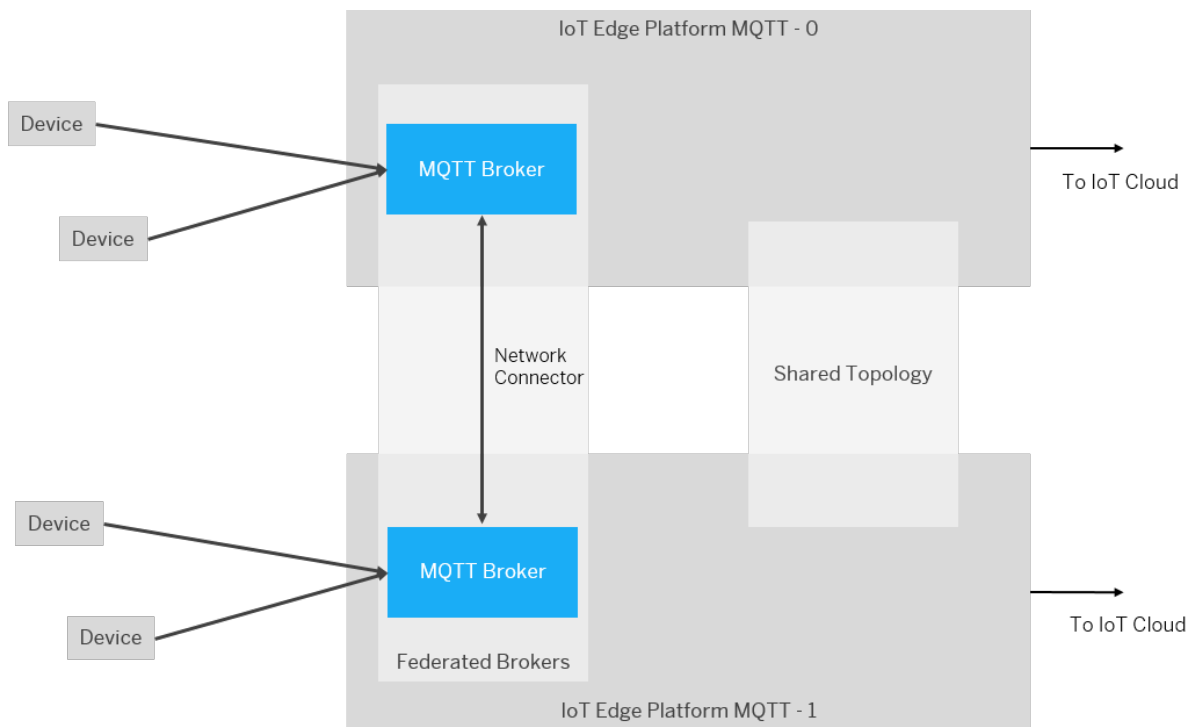
## 5.1.2 Generic Aspects

From a functional perspective, a Internet of Things Edge Platform scale-out deployment targets the following goals:

- All Internet of Things Edge Platform instances in the cluster are associated to the same platform-generated unique ID, hence, from your perspective, the cluster behaves like a single Internet of Things Edge Platform. Internet of Things Edge Platform instances within the cluster must implement the same protocol.
- Each instance in the cluster exposes a protocol endpoint, for example a REST service endpoint or an MQTT transport. Therefore, a cluster of N instances comprises N protocol endpoints overall. From a device perspective, the cluster provides the same functionality regardless of the endpoint the device connects to.
- The Internet of Things Edge Platform instances share the overall ingestion load produced by devices (uplink flow).
- Device commands are managed by only one Internet of Things Edge Platform instance at a time; any Internet of Things Edge Platform instance in the cluster is capable of processing the message successfully (downlink flow).
- Device model entities (device, sensor) created by an Internet of Things Edge Platform instance must be known by each instance in the cluster. This is achieved by sharing the topology information that describes the managed device model entities across the cluster.

## 5.1.3 MQTT Overview

The following picture provides an overview of a cluster of MQTT Internet of Things Edge Platform instances:



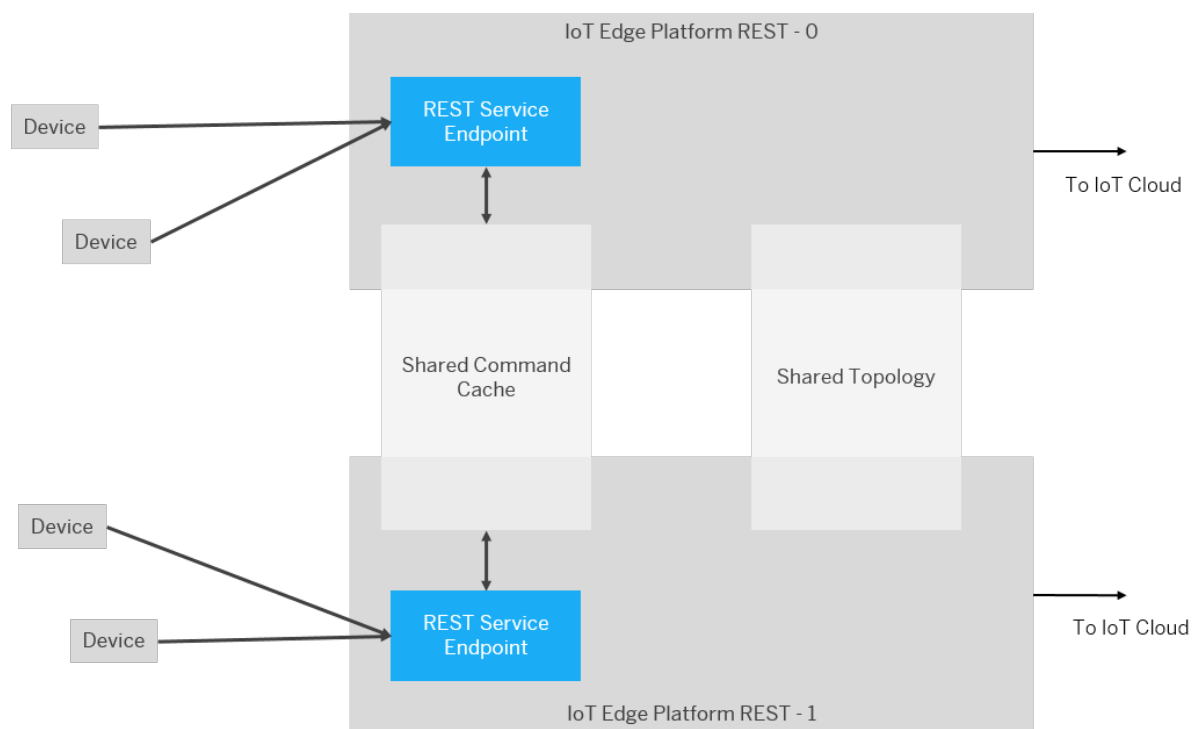
Two elements represent the foundation of the scale-out deployment for MQTT:

- The shared topology layer
- The configuration of the MQTT brokers within each Internet of Things Edge Platform instance as a cluster of federated brokers

In the context of the downlink communication with devices, federating the MQTT brokers is required to share device subscription metadata across all clustered brokers. This allows a device to consume commands regardless of the broker it connects to. On the uplink path, message dispatch must be appropriately configured to ensure that only one Internet of Things Edge Platform instance within the cluster receives the message. This is required to avoid pushing a duplicated device measurement to other components on the pipeline.

## 5.1.4 REST Overview

The following picture provides an overview of a cluster of REST Internet of Things Edge Platform instances:



Two elements represent the foundation of the cluster deployment for Internet of Things Edge Platform REST:

- The shared topology layer.
- The shared commands cache, to make commands available to all instances in the cluster.

## 5.2 Internet of Things Edge Platform Configuration

This section contains the guidelines to configure the Internet of Things Edge Platform instances in a scale-out deploy. The scale-out configuration is applied on top of the standard configuration as described in the tutorials [Set Up the Internet of Things Edge Platform MQTT](#) and [Set Up the Internet of Things Edge Platform REST](#).

Scale-out configuration guidelines are provided in a tabular manner and include examples to clarify their scope and content. For the reader's convenience, guidelines are also grouped in categories, identified by letters:

- Generic configuration, that applies to both the MQTT and REST adapters:
  - Gateway alternateId configuration (category **A**)
  - Shared cache configuration (category **B**)
  - Scale-out enablement (category **C**)
- MQTT-specific configuration (category **D**)
- REST-specific configuration.

Within each category, guidelines are identified with a progressive number (for example **A1**, **A2**). Guideline categories **A**, **B**, **C**, **D** apply for the MQTT adapter, whereas only guidelines **A**, **B**, **C** apply for REST adapter (that doesn't require any specific configuration as of today). In some cases, a certain configuration item is addressed in alternative ways: In such a case, two different guidelines are given, identified with same category and progressive number and differentiated by a third lower-case letter (for example **B1a**, **B1b**). It is under the reader's responsibility to choose which of the alternative guidelines to apply, based on the clarifications provided in the guidelines description.

## Generic Configuration

This section details the configuration aspects common to the MQTT and the REST use cases. Throughout the section, the following conventional identifiers are used:

- `<iot_edge_platform_folder>`: Identifies the folder on the target host where a certain Internet of Things Edge Platform instance is installed.
- `<adapter_protocol>`: Identifies the protocol the Internet of Things Edge Platform instance is managing. It can be either `mqtt` or `rest`.

## Gateway alternateId Configuration

The same Gateway `alternateId` must be set on all Internet of Things Edge Platform within the cluster.

### A1:

Open the `config_gateway_<adapter_protocol>.xml` file (available in the `<iot_edge_platform_folder>/config` folder) and add the `gatewayAlternateId` attribute to the xml element `<cnf:gateway>`. Set the desired value for the attribute.

### Example:

#### Sample Code

```
<cnf:gateway gatewayAlternateId="GatewayMQTT_TEST">
```

### A2:

Open the `hazelcast-group.xml` file (available in the `<gateway_deploy_folder>/config` folder) and replace the placeholder `gatewayAlternateId` in the name of the group with the real `alternateId` of the Gateway, as inserted at the previous point.

### Example:

#### Sample Code

```
<group>
  <name>iotservices-gateway-GatewayMQTT_TEST</name>
  <password>iotservices-pass</password>
</group>
```

The chosen gateway `alternateId` value must be unique at Internet of Things Service tenant level. Other Internet of Things Edge Platform instances not included in the same scale-out deploy must have a different gateway `alternateId`.

## Shared Cache Configuration

Internet of Things Edge Platform instances in the same cluster share device topology information through a shared Hazelcast cache. In case Internet of Things Edge Platform implements the REST adapter, the same technology backs up a shared command cache. This section details the Hazelcast configuration.

### B1 - Hazelcast Network:

Open the `hazelcast.xml` file (available in the `<iot_edge_platform_folder>/config` folder) and edit the `<network>/<port>` section to configure the port used by Hazelcast. Two possibilities are available:

- Look for the first available port in a given range, starting from a given port value. This can be achieved through the `auto-increment` option: Set the `auto-increment` attribute to `true` and specify the range size with the `port-count` attribute. The starting port number is determined by the value of the `<port>` tag.
- Use a fixed port value: Set the `auto-increment` attribute to `false` and specify the port number as value of the `<port>` tag.

In case some Internet of Things Edge Platform instances run on the same host, pay attention to enforce a port configuration that guarantees the binding of each instance to a free port. In such a context, enabling the `auto-increment` feature allows you to configure the network in the same way on all Internet of Things Edge Platform instances. Hazelcast automatically allocates different ports for the instances. In case `auto-increment` is turned off, you are responsible for setting the fix port value appropriately on the Internet of Things Edge Platform instances running on the same host.

**Example:** To start from the port 6700 and use the first available one (hopefully the same 6700) for your Hazelcast network, edit the file as follows:

#### Sample Code

```
<port auto-increment="true" port-count="100">6700</port>
```

A fixed port can be used by disabling the "auto-increment" feature:

#### Sample Code

```
<port auto-increment="false">6700</port>
```

## Join Strategies

Hazelcast provides different strategies to join members in the group. Multicast represents the easiest way to scale out, as new Internet of Things Edge Platform instances join the Hazelcast group dynamically. But you can opt as well for a static group configuration, in order to have a scale-out setup where new members join in a controlled way.

The following subsections explain how to enable such two alternative strategies.



## B2a - Multicast

To enable the multicast join, edit the `hazelcast.xml` file (available in the `<iot_edge_platform_folder>/config` folder), setting the element `<multicast enabled="true">` inside the `<join>` section. Set an address and a port for the multicast and make sure that they are accessible.

Configure the multicast in the same way for all the gateways you want to group together. Other elements inside the `<join>` section must be disabled. For example, disable the `<tcp-ip>` (`<tcp-ip enabled="false">`) and the `<aws>` (`<aws enabled="false">`) elements).

### Example:

#### Sample Code

```
...
    </outbound-ports>
    <join>
      <multicast enabled="true">
        <multicast-group>224.2.2.3</multicast-group>
        <multicast-port>54327</multicast-port>
      </multicast>
    ...
```

## B2b - Static Groups

To enable the tcp-ip static join, edit the `hazelcast.xml` file (available in the `<iot_edge_platform_folder>/config` folder), setting the element `<tcp-ip enabled="true">` inside the `<join>` section. Set the right address for the interface, and configure the member list adding all the members (expressed in the `address: port` format) in the cluster.

### Example:

The following configuration groups together Internet of Things Edge Platform X and Y, both running on the same machine (see the configuration of the IP address to localhost):

- Internet of Things Edge Platform X uses port 6700 and looks for a member on port 6701.

#### Example:

#### Sample Code

```
<tcp-ip enabled="true">
  <interface>127.0.0.1</interface>
  <member-list>
    <member>127.0.0.1:6701</member>
  </member-list>
</tcp-ip>
```

- Internet of Things Edge Platform Y uses port 6701 and looks for a member on port 6700.

#### Example:

#### Sample Code

```
<tcp-ip enabled="true">
  <interface>127.0.0.1</interface>
  <member-list>
    <member>127.0.0.1:6700</member>
  </member-list>
</tcp-ip>
```

## Scale-Out Enablement

The support for a scale-out deploy must be activated on Internet of Things Edge Platform by setting a dedicated configuration parameter passed as a VM argument.

### C1:

Edit the `gateway.bat/.sh` file (available in the `<iot_edge_platform_folder>` folder), adding the `VMArg com.sap.iotservices.gateway.enableScalability` and setting it to `true`.

### Example:

#### Sample Code

```
-Dcom.sap.iotservices.gateway.enableScalability=true
```

## 5.3 MQTT-Specific Configuration

As mentioned, the MQTT brokers within each clustered Internet of Things Edge Platform instance must be federated. This can be achieved by configuring a network connector that brokers leverage to share metadata and messages. As the network connector establishes a bidirectional channel between two brokers, it doesn't need to be set in all configuration files of all Internet of Things Edge Platform instances in the cluster. The network connector configuration includes an additional node (`cnf:excludedDestinations`) that regulates the forwarding on the `measures` topic, ensuring that measurement messages are processed by only one Internet of Things Edge Platform instance in the cluster. This avoids the forwarding of duplicated measurements in the Internet of Things Service pipeline.

### D1:

Edit the `config_gateway_mqtt.xml` file (available in the `<iot_edge_platform_folder>/config` folder) and declare the network connector in the `<cnf:gatewayBundle>/<cnf:amq>` section. In case of two clustered Internet of Things Edge Platform instances, the network connector must be configured only on one of them (be it instance **X**) and must point to the NIO+SSL port of the other (be it instance **Y**). Don't configure network connectors on instance **Y**. In the more generic case of N brokers in the cluster:

- The first one to be started has no `networkConnectors` configuration.
- Each new broker that joins the cluster needs to have configured `networkConnectors` pointing to the previously started brokers.

### Example:

Considering two Internet of Things Edge Platform instances **X** and **Y** with the following parameters:

- `<instance_x_address>`: The IP address for instance **X**
- `<instance_y_address>`: The IP address for instance **Y**
- `<instance_x_port>`: The NIO+SSL port for instance **X** (that must be specified in the `transportConnector` named "nio+ssl" in the `<cnf:amq>/<cnf:connectors>/<cnf:server>/<cnf:transportConnectors>` node of the `config_gateway_mqtt.xml` file (available in the `<iot_edge_platform_folder>/config` folder))

- `<instance_Y_port>`: The NIO+SSL port for instance **Y**

Assuming that instance **Y** is started first, the `networkConnector` must be specified only in instance **X** configuration file.

Configuration for instance **X**:

#### Sample Code

```
<cnf:amq>
...
<cnf:connectors>
  <cnf:server>
    <cnf:networkConnectors>
      <cnf:networkConnector uri="static:(ssl://
<instance_Y_address>:<instance_Y_port>?verifyHostName=false)" networkTTL="1"
duplex="true">
        <cnf:excludedDestinations>
          <cnf:dest topic="measures."/>
        </cnf:excludedDestinations>
      </cnf:networkConnector >
    </cnf:networkConnectors>
    <cnf:transportConnectors>
      ...
      <cnf:transportConnector
name="nio+ssl" uri="nio+ssl://<instance_X_address>:<instance_X_port>?
needClientAuth=true&transport.soTimeout=60000"/>
      ...
    </cnf:transportConnectors>
    ...
  </cnf:server>
</cnf:connectors>
</cnf:amq>
```

Configuration for instance **Y**:

#### Sample Code

```
<cnf:amq>
...
<cnf:connectors>
  <cnf:server>
    <cnf:transportConnectors>
      ...
      <cnf:transportConnector
name="nio+ssl" uri="nio+ssl://<instance_Y_address>:<instance_Y_port>?
needClientAuth=true&transport.soTimeout=60000"/>
      ...
    </cnf:transportConnectors>
    ...
  </cnf:server>
</cnf:connectors>
</cnf:amq>
```

#### Example:

Assuming a scale-out setup with four Internet of Things Edge Platform instances (**X**, **Y**, **Z**, **T**), the configuration of the fourth one (**T**) must have connectors pointing to the other three brokers previously started:

#### Sample Code

```
<cnf:networkConnectors>
```

```

    <cnf:networkConnector uri="static:
(ssl://<instance_X_address>:<instance_X_port>?verifyHostName=false,ssl://
<instance_Y_address>:<instance_Y_port>?verifyHostName=false,ssl://
<instance_Z_address>:<instance_Z_port>?verifyHostName=false)" networkTTL="1"
duplex="true">
      <cnf:excludedDestinations>
        <cnf:dest topic="measures."> />
      </cnf:excludedDestinations>
    </cnf:networkConnector>
  </cnf:networkConnectors>

```

## D2:

Each broker in the cluster must have a unique name. It's advised to follow the convention `gateway-mqtt-<broker-instance-index>`, where `broker-instance-index` is an integer number incremented whenever a new instance joins the cluster.

### Example:

Considering two Internet of Things Edge Platform instances **X** and **Y**.

Configuration for instance **X**:

#### Sample Code

```

<cnf:amq>
    <cnf:brokerName>gateway-mqtt-1</cnf:brokerName>
    <cnf:connectors>
      ...
    </cnf:connectors>
</cnf:amq>
<cnf:gateway>
    ...
    <cnf:gateway_comm>
      ...
      <cnf:jmsConnectionString>failover:(vm://gateway-mqtt-1?
waitForStart=30000&create=false)</cnf:jmsConnectionString>
      ...
    </cnf:gateway_comm>
    ...
</cnf:gateway>

```

Configuration for instance **Y**:

#### Sample Code

```

<cnf:amq>
    <cnf:brokerName>gateway-mqtt-0</cnf:brokerName>
    <cnf:connectors>
      ...
    </cnf:connectors>
</cnf:amq>
<cnf:gateway>
    ...
    <cnf:gateway_comm>
      ...
      <cnf:jmsConnectionString>failover:(vm://gateway-mqtt-0?
waitForStart=30000&create=false)</cnf:jmsConnectionString>
      ...
    </cnf:gateway_comm>
    ...
</cnf:gateway>

```

Apart from the cluster-specific configuration described above, each Internet of Things Edge Platform instance must adhere to the configuration guidelines that generally apply to the MQTT adapter. For more information, please refer to the tutorial [Set Up the Internet of Things Edge Platform MQTT](#).

## 5.4 REST-Specific Configuration

No specific configuration is required to enable a scale-out setup for Internet of Things Edge Platform REST. Each instance must adhere to the configuration guidelines that generally apply to the REST adapter. For more information, please refer to the tutorial [Set Up the Internet of Things Edge Platform REST](#).

## 5.5 Deployment of Custom Bundles

As mentioned previously, the Internet of Things Edge Platform scale-out deploy doesn't support the management of custom OSGi bundles through the `/tenant/{tenantId}/gateways/{gatewayId}/bundles` APIs. Though, custom OSGi bundles can still be installed manually in each Internet of Things Edge Platform within the scale-out deploy. This section provides the guidelines for the bundle manual installation. The guidelines are grouped in categories and laid out in a tabular manner, including examples to clarify the information provided.

The procedure below allows you to install a single OSGi bundle on an Internet of Things Edge Platform instance. It must be repeated for all Internet of Things Edge Platform instances in the scale-out deploy and for all custom bundles.

### Adding the Bundle File

The custom OSGi bundle is provided as a `*.jar` file. It must be stored in the `custombundles` folder under the Internet of Things Edge Platform deploy folder.

#### E1:

If not already present, create a folder `custombundles` under the folder `<iot_edge_platform_folder>`. Then add the bundle file to it.

### Modifying the config.ini File

The `config.ini` file regulates the startup order of the OSGi bundles that make up the Internet of Things Edge Platform. Any bundle that must be started must be included in this file. The file contains a number of

properties that control the behaviour of the OSGi runtime; the list of bundles to be managed is specified under property `osgi.bundles`.

### E2:

Edit the `config.ini` file (available in the `<iot_edge_platform_folder>/configuration` folder), adding a new line for each bundle to be added.

#### Note

Create a backup copy of this file before editing it.

### Example:

Considering the following parameters:

- `<bundle-file>`: Represents the OSGi bundle, packaged as a `*.jar` file.
- `<bundle-start-level>`: Represents the start level for the custom OSGi bundle. It must be set to 6 to start the custom bundle after the other platform bundles are started.

The new line would then be: `custombundles/<bundle-file>@<bundle-start-level>:start, \`

The line is added in the second to last position, before the line related to the adapter bundle: for example in MQTT, the end of the list would look like:

#### Sample Code

```
..plugins/parser-utils-service@3:start, \
custombundles/<bundle-file>@<bundle-start-level>:start, \
plugins/gateway-mqtt@start
```

Make sure that the file indentation is respected when adding the new line by adding an equivalent number of space characters at its beginning. Do the same for the other entries in the bundles' list.

## Adding the custom\_config.ini File

The `custom_config.ini` file contains the subset of lines within the `config.ini` file that relates to custom bundles. It's used to identify the custom bundles when upgrading the Internet of Things Edge Platform.

### E3:

Open the folder `custombundles` under folder `<iot_edge_platform_folder>` and create a text file named `custom_config.ini`, if not already present. Then open the `config.ini` file, identify the line related to the target bundle, and copy it into the `custom_config.ini` file (including all starting space characters).

## 5.6 Internet of Things Edge Platform Upgrade

In general, the upgrade of an Internet of Things Edge Platform instance in a scale-out deploy is performed as the upgrade of a single-instance deploy. You must retrieve the `*.zip` archive containing the new Internet of

Things Edge Platform version and set up each instance following the instructions in the tutorials [Set Up the Internet of Things Edge Platform MQTT](#) or [Set Up the Internet of Things Edge Platform REST](#).

If necessary, porting the custom OSGi bundles can be automated by running the `build.sh` or `build.bat` script with a parameter pointing to the deploy folder of the current Internet of Things Edge Platform instance.

As a first step, all running instances within the scale-out deploy must be stopped. Then, you can proceed with the upgrade of each single instance:

- Unzip the archive of the new Internet of Things Edge Platform instance into a folder (for example `<iot_edge_platform_new_version_folder>`).
- Enter the `<iot_edge_platform_new_version_folder>` directory.
- Run the build script passing the path of the current Internet of Things Edge Platform instance, for example `<iot_edge_platform_old_version_folder>`:

#### Sample Code

```
./build.sh <adapter_protocol> <iot_edge_platform_old_version_folder>
```

or

#### Sample Code

```
build.bat <adapter_protocol> <iot_edge_platform_old_version_folder>
```

- Configure the instance as described in the tutorials [Set Up the Internet of Things Edge Platform MQTT](#), [Set Up the Internet of Things Edge Platform REST](#), copying the values from the current instance deploy.
- Apply the scale-out configuration described in the previous sections, excluding the steps related to the installation of the custom OSGi bundles. Copying the configuration values from the current instance deploy.
- Start the instance.

The upgraded instance can be either started with a tenant registration certificate or with the unique gateway certificate retrieved by the current instance at its first startup. In the latter case, you must manually copy the `certificate` folder from the old instance (folder `<iot_edge_platform_old_version_folder>`) into the new one (`<iot_edge_platform_new_version_folder>`).

## 5.7 Internet of Things Edge Platform Startup

The Internet of Things Edge Platform startup must be executed for each instance in the cluster as described in the tutorials [Set Up the Internet of Things Edge Platform MQTT](#) or [Set Up the Internet of Things Edge Platform REST](#).

## 5.8 Networking Setup

As described previously, an Internet of Things Edge Platform scale-out deploy provides devices with as many protocol endpoints as the number of instances in the cluster. The cluster exposes the same functionality regardless of the specific endpoint the device connects to. A TCP/HTTP load-balancing layer represents an attractive solution to effectively share incoming connections across all of the available MQTT/REST endpoints. As Internet of Things Edge Platform doesn't provide any load-balancing functionality, you must provide it as part of the networking infrastructure the cluster fits in.





# Important Disclaimers and Legal Information

## Hyperlinks

Some links are classified by an icon and/or a mouseover text. These links provide additional information.

About the icons:

- Links with the icon : You are entering a Web site that is not hosted by SAP. By using such links, you agree (unless expressly stated otherwise in your agreements with SAP) to this:
  - The content of the linked-to site is not SAP documentation. You may not infer any product claims against SAP based on this information.
  - SAP does not agree or disagree with the content on the linked-to site, nor does SAP warrant the availability and correctness. SAP shall not be liable for any damages caused by the use of such content unless damages have been caused by SAP's gross negligence or willful misconduct.
- Links with the icon : You are leaving the documentation for that particular SAP product or service and are entering an SAP-hosted Web site. By using such links, you agree that (unless expressly stated otherwise in your agreements with SAP) you may not infer any product claims against SAP based on this information.

## Videos Hosted on External Platforms

Some videos may point to third-party video hosting platforms. SAP cannot guarantee the future availability of videos stored on these platforms. Furthermore, any advertisements or other content hosted on these platforms (for example, suggested videos or by navigating to other videos hosted on the same site), are not within the control or responsibility of SAP.

## Beta and Other Experimental Features

Experimental features are not part of the officially delivered scope that SAP guarantees for future releases. This means that experimental features may be changed by SAP at any time for any reason without notice. Experimental features are not for productive use. You may not demonstrate, test, examine, evaluate or otherwise use the experimental features in a live operating environment or with data that has not been sufficiently backed up.

The purpose of experimental features is to get feedback early on, allowing customers and partners to influence the future product accordingly. By providing your feedback (e.g. in the SAP Community), you accept that intellectual property rights of the contributions or derivative works shall remain the exclusive property of SAP.

## Example Code

Any software coding and/or code snippets are examples. They are not for productive use. The example code is only intended to better explain and visualize the syntax and phrasing rules. SAP does not warrant the correctness and completeness of the example code. SAP shall not be liable for errors or damages caused by the use of example code unless damages have been caused by SAP's gross negligence or willful misconduct.

## Bias-Free Language

SAP supports a culture of diversity and inclusion. Whenever possible, we use unbiased language in our documentation to refer to people of all cultures, ethnicities, genders, and abilities.

© 2024 SAP SE or an SAP affiliate company. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP SE or an SAP affiliate company. The information contained herein may be changed without prior notice.

Some software products marketed by SAP SE and its distributors contain proprietary software components of other software vendors. National product specifications may vary.

These materials are provided by SAP SE or an SAP affiliate company for informational purposes only, without representation or warranty of any kind, and SAP or its affiliated companies shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP or SAP affiliate company products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP SE (or an SAP affiliate company) in Germany and other countries. All other product and service names mentioned are the trademarks of their respective companies.

Please see <https://www.sap.com/about/legal/trademark.html> for additional trademark information and notices.