IBM IOT Foundation Documentation

Release 1.0

David Parker

June 09, 2015

Contents

CHAPTER 1

About

Welcome to the documentation/reference material for IBM's IOT Foundation service. If you are looking for tutorials you can find them on DeveloperWorks

The documentation is organized into a number of sections:

CHAPTER 2

Reference Material

2.1 Concepts

2.1.1 Organizations

When you register with the Internet of Things Foundation you are given an organization ID, this is a unique 6 character identifier for your account.

Organizations ensure your data is only accessible from your devices and applications. Once registered, devices and API keys are bound to a single organization. When an application connects to the service using an API key it registers with the organization that "owns" the API key.

For your security it is impossible for cross-organization communication within the Internet of Things Foundation eco-system, intentional or otherwise. The only way to transmit data between two organizations it to explicitly create two applications, one within each organization, that communicate with each other to act as a relay between the organizations.

2.1.2 Devices

- A device can be anything that has a connection to the internet and has data it wants to get into the cloud.
- A device is not able to directly interact with other devices.
- Devices are able to accept commands from applications.
- Devices uniquely identify themselves to the IoT Foundation with an authentication token that will only be accepted for that device.
- Devices must be registered before they can connect to the IoT Foundation.

2.1.3 Applications

- An application is anything that has a connection to the internet and wants to interact with data from devices and/or control the behaviour of those devices in some manner.
- Applications identify themselves to the IoT Foundation with an API key and a unique application ID.
- Applications do not need to be registered before they can connect to the IoT Foundation, however they must present a valid API key that has previously been registered.

2.1.4 Events

Events are the mechanism by which **devices** publish data to the Internet of Things Foundation. The device controls the content of the event and assigns a name for each event it sends.

When an event is received by the IOT Foundation the credentials of the connection on which the event was received are used to determine from which device the event was sent. With this architecture it is impossible for a device to impersonate another device.

Devices are unable to receive events, regardless of whether they are its own events or those of another device.

Applications are able to process events from devices in real time. When an application receives an event it has visibility of the source of the event and the data contained in that event. Applications can be configured to subscribe to all events from all devices, a subset of events, a subset of devices or a combination of these events.

2.1.5 Commands

Commands are the mechanism by which **applications** can communicate with **devices**. Only applications can send commands, which must be issued to specific devices.

The device must determine which action to take on receipt of any given command, and even controls which commands it will subscribe to in the first place. It is possible to design your device to listen for any command, or to simply subscribe to a set of specific commands.

2.2 Building the Internet of Things

The IBM Internet of Things Foundation is powered by a core of IBM's leading products and services:

- IBM DataPower Gateway
- IBM WebSphere Application Server Liberty Core
- IBM Informix TimeSeries
- IBM MessageSight
- Cloudant
- SoftLayer

The Foundation is also built upon a number of open source projects, including:

- Logging
  - Logstash
  - Redis
  - Elasticsearch
  - Kibana
- Service Discovery
  - Consul
- Continuous Delivery
  - CasperJS
  - Chef

  - Docker
  - Karma
  - Mocha
- Statistics & Monitoring
  - Graphite
  - Diamond
  - StatsD
  - Grafana
- User Interface
  - AngularJS
  - Bootstrap

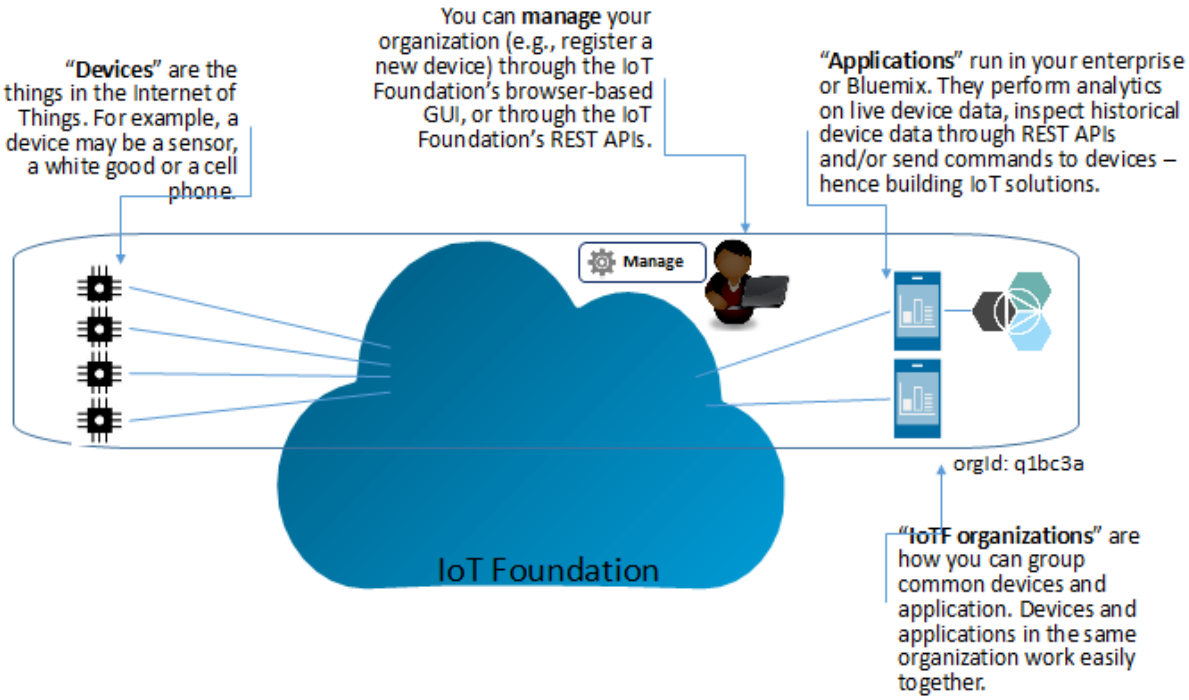
2.3 Securing the Internet of Things

The IBM Internet of Things Foundation (IoT Foundation) is a fully managed, cloud-hosted service that makes it simple to derive value from Internet of Things (IoT) devices.

The following document aims to answer common questions about how your organization's data is protected. Focusing on specific areas:

- Authentication: assuring the identity of users, devices or applications attempting to access your organization's information.
- Authorization: assuring that users, devices and applications have permission to access your organization's information.
- Encryption: assuring that data is only readable by authorized parties and cannot be intercepted.

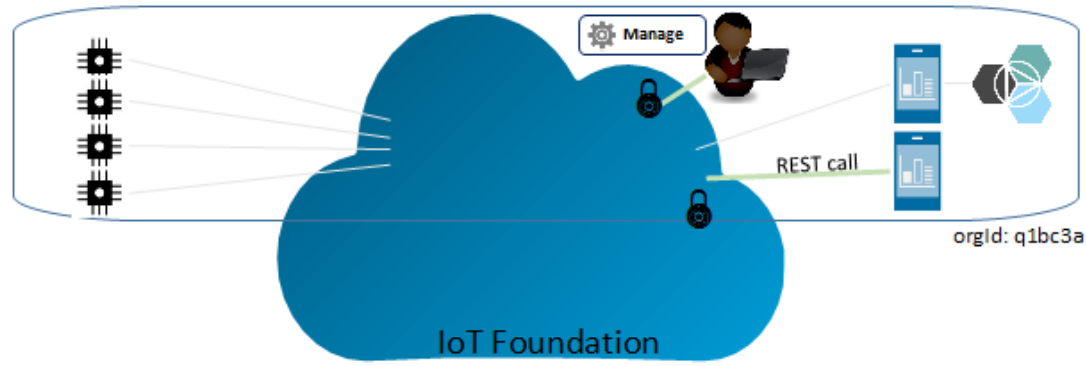
2.3.1 Terminology



2.3.2 How do we secure management of your organization?

The browser-based GUI and REST APIs are fronted by HTTPS, with a certificate signed by DigiCert enabling you to trust that you're connecting to the genuine IoT Foundation.

- GUI: authenticated via your IBM ID.
- REST API: once you create an API key through the GUI, you can use this to make authenticated REST calls against your organization.

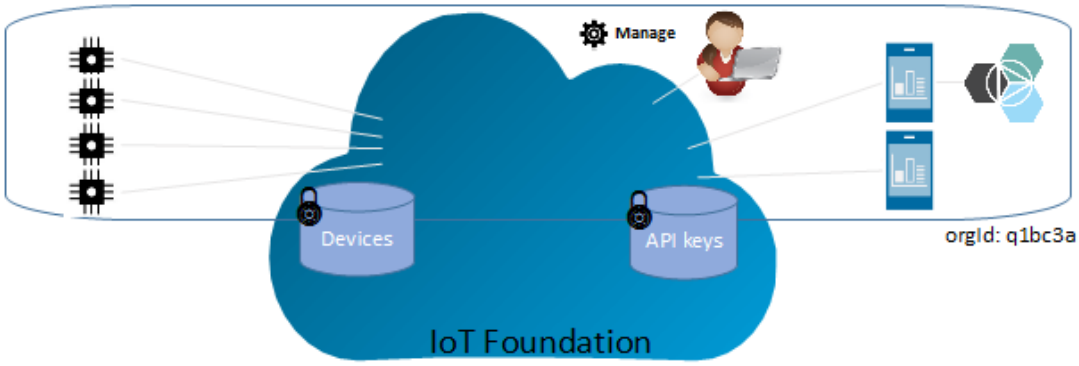


2.3.3 How do we secure your device and application credentials?

When devices are registered or API keys are generated, the authentication token is salted and hashed. This means your organization's credentials can never be recovered from our systems - even in the unlikely event that the IoT Foundation

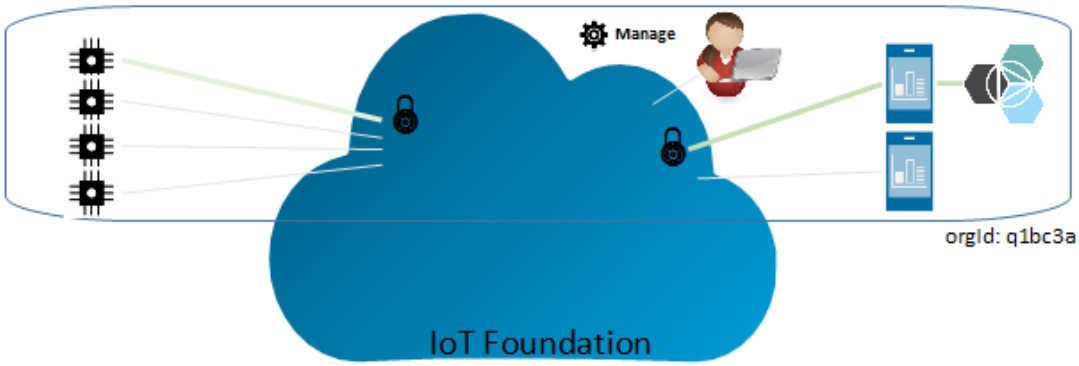
is compromised.

Device credentials and API keys can be individually revoked if they are compromised.



2.3.4 How do we ensure your devices connect securely to the IoT Foundation?

- Devices connect through a unique combination of clientId and authentication token that only you know.
- Full support for connectivity over TLS (v1.2) is provided.
- Open standards are used (MQTT v3.1.1) to allow easy interop across many platforms and languages.



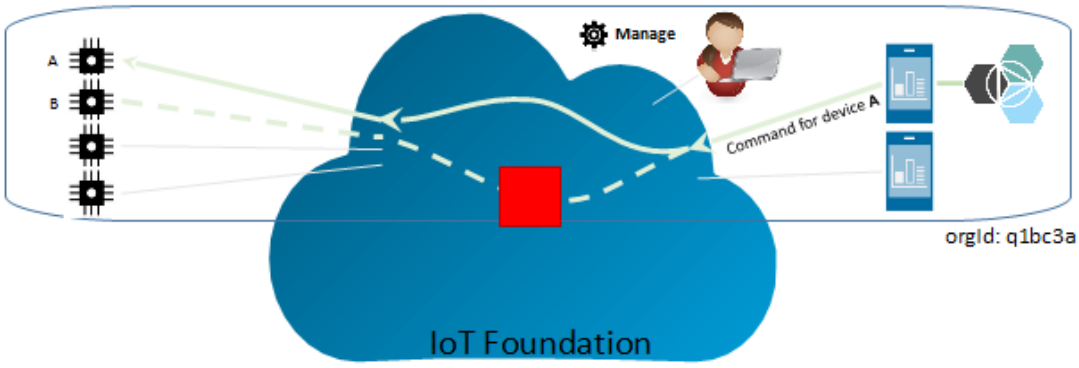
2.3.5 How do we prevent data leaking between devices?

Secure messaging patterns are baked in. Once authenticated, devices are only authorized to publish and subscribe to a restricted topic space:

- /iot-2/evt/+/fmt/+
- /iot-2/cmd/+

All devices work with the same topic space. The authentication credentials provided by the client connecting dictate to which device this topic space will be scoped by the IOT Foundation. This prevents devices from being able to impersonate another device.

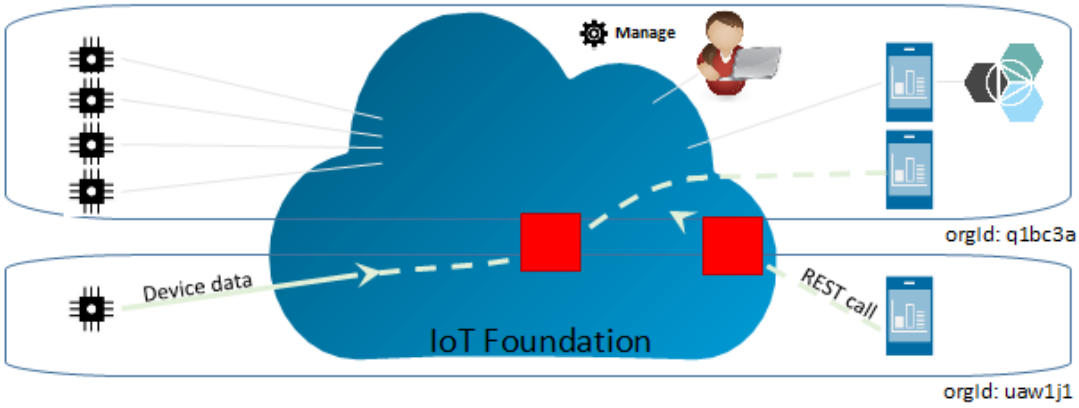
The only way to impersonate another device is by obtaining compromised security credentials for the device.



Applications can subscribe and publish on both the event and command topics for all devices in the organization. Applications can analyse data from may devices simultaneously, and can also simulate and proxy devices in addition to forming the complementary side of a full duplex communication loop.

2.3.6 How do we prevent data leaking between organizations?

The topic space in which devices and applications operate is scoped within a single organization. When authenticated, the IoT Foundation transforms the topic structure using an organization ID based on the client authentication, making it impossible for data from one organization to be accessed from another.



CHAPTER 3

API

3.1 The Device Management API

3.1.1 Authentication

The API is only accessible over HTTPS and is protected by HTTP Basic Authentication, you must provide a username and password with every request.

Username

A valid API key registered to the organization you want to work with.

Password

The authentication token corresponding to your API key.

3.1.2 View Organization

Retrieve details about an organization.

Request

GET **org_id**.internetofthings.ibmcloud.com/api/v0001/

Request Body

None

Response Body

- id - String
- name - String
- created - ISO8601 date string
- updated - ISO8601 date string

Example Response

```
{
  "id": "ey67sp",
  "name": "My Organization",
  "created":"2014-07-04T21:36:05Z",
  "updated":"2014-07-04T21:36:05Z"
}
```

Common Response Codes

- 200 (Success)

3.1.3 List Devices

Get a list of all devices in an organization.

Request

GET **org_id**.internetofthings.ibmcloud.com/api/v0001/devices

Request Body

None

Response Body

The response body will contain a list of devices, as below:

- uuid - String
- type - String
- id - String
- metadata - Complex data (if available)
- registration
  - auth
    - id - String

        * type - String
    - date - ISO8601 date string

Example Response


```
[
  {
    "uuid": "d:ey67sp:raspberrypi-sample:1958138a4dfe",
    "type": "raspberrypi-sample",
    "id": "1958138a4dfe",
    "metadata": {
      "address": {
        "number": 29,
        "street": "Acacia Road"
      }
    },
    "registration": {
      "auth": {
        "id": "joebloggs@uk.ibm.com",
        "type": "person"
      },
      "date": "2014-08-21T18:25:43-05:00"
    }
  },
  {
    "uuid": "d:ey67sp:mbed-sample:1253138b4dcd",
    "type": "mbed-sample",
    "id": "1253138b4dcd",
    "metadata": {
      "address": {
        "number": 13,
        "street": "Elm Street"
      }
    },
    "registration": {
      "auth": {
        "id": "joebloggs@uk.ibm.com",
        "type": "person"
      },
      "date": "2014-08-21T18:25:43-05:00"
    }
  }
]
```


Common Response Codes

- 200 (Success)

3.1.4 List Devices by Type

Get a list of all devices of a specific type in an organization.

Request

GET **org_id**.internetofthings.ibmcloud.com/api/v0001/devices/**device_type**

Request Body

None

Response Body

The response body will contain a list of devices, as below:

- uuid - String
- type - String
- id - String
- metadata - Complex data
- registration
  - auth
    - id - String
    - type - String
  - date - ISO8601 date string

Example Response

```
[
  {
    "uuid": "d:ey67sp:raspberrypi-sample:1958138a4dfe",
    "type": "raspberrypi-sample",
    "id": "1958138a4dfe",
    "metadata": {
      "address": {
        "number": 29,
        "street": "Acacia Road"
      }
    },
    "registration": {
      "auth": {
        "id": "joebloggs@uk.ibm.com",
        "type": "person"
      },
      "date": "2014-08-21T18:25:43-05:00"
    }
  }
]
```

Common Response Codes

- 200 (Success)

3.1.5 List Device Types

Get a list of all devices types in an organization.

Request

GET **org_id**.internetofthings.ibmcloud.com/api/v0001/device-types

Request Body

None

Response Body

The response body will contain a list of device types:

- deviceType - String
- count - Integer

Example Response

```
[
  {
    "deviceType": "raspberrypi-sample",
    "count": 1
  },
  {
    "deviceType": "mbed-sample",
    "count": 1
  }
]
```

Common Response Codes

- 200 (Success)

3.1.6 Register a New Device

Register a new device to an organization.

Note: You can use any scheme of your choice when assigning values for type and id to registered devices, however the following restrictions apply:

- Maximum length of 32 characters
- Must comprise only alpha-numeric characters and the following special characters:
  - dash ("-")
  - underscore ("_")

- dot (".")

Request

POST **org_id**.internetofthings.ibmcloud.com/api/v0001/devices

Request Body

You must specify the type and identifer of the device being registered.

- type - String
- id - String
- metadata - Complex data (optional)

Example Request

```
{
  "type": "raspberrypi-sample",
  "id": "1958138a4dfe",
  "metadata": {
    "address": {
      "number": 29,
      "street": "Acacia Road"
    }
  }
}
```

Response Headers

The response header will contain the location (resource URI) for the registered device. * Location - URI

Response Body

The response body will contain a uuid and password for the registered device.

- uuid - String
- type - String
- id - String
- metadata - Complex data (if available)
- password - String

Important: The response body will contain the generated authentication token for this device. You must make sure to record this token when processing the response. The data is stored hashed and salted so we are not able to retrieve lost authentication tokens.

Example Response


```
{
  "uuid": "d:ey67sp:raspberrypi-sample:1958138a4dfe",
  "type": "raspberrypi-sample",
  "id": "1958138a4dfe",
  "metadata": {
    "address": {
      "number": 29,
      "street": "Acacia Road"
    }
  },
  "password": "A?j8y_ueh*d(je34",
  "registration": {
    "auth": {
      "id": "joebloggs@uk.ibm.com",
      "type": "person"
    },
    "date": "2014-08-21T18:25:43-05:00"
  }
}
```


Common Response Codes

- 201 (Created) - The device was successfully registered (Location header set to the URL of the new device)

3.1.7 Update a Registered Device

Update an existing device in an organization.

Request

PUT **org_id**.internetofthings.ibmcloud.com/api/v0001/devices/**device_type/device_id**

Request Body

Currently, the only property of a device that can be changed after initial registration is its metadata.

- metadata - Complex data

Example Request


```
{
  "metadata": {
    "address": {
      "number": 21,
      "street": "Acacia Avenue"
    }
  }
}
```

Response Body

The response body will contain the updated properties of the registered device.

- uuid - String
- type - String
- id - String
- metadata - Complex data
- registration
  - auth
    - id - String
    - type - String
  - date - ISO8601 date string

Example Response

```
{
  "uuid": "d:ey67sp:raspberrypi-sample:1958138a4dfe",
  "type": "raspberrypi-sample",
  "id": "1958138a4dfe",
  "metadata": {
    "address": {
      "number": 21,
      "street": "Acacia Avenue"
    }
  },
  "registration": {
    "auth": {
      "id": "joebloggs@uk.ibm.com",
      "type": "person"
    },
    "date": "2014-08-21T18:25:43-05:00"
  }
}
```

Common Response Codes

- 200 (Success)

3.1.8 View a Registered Device

Get summary information about a registered device in an organization.

Request

GET **org_id**.internetofthings.ibmcloud.com/api/v0001/devices/**device_type/device_id**

Request Body

None

Response Body

The response body will contain the known properties of the device.

- uuid - String
- type - String
- id - String
- metadata - Complex data
- registration
  - auth
    - id - String
    - type - String
  - date - ISO8601 date string

Example Response

```
{
  "uuid": "d:ey67sp:raspberrypi-sample:1958138a4dfe",
  "type": "raspberrypi-sample",
  "id": "1958138a4dfe",
  "metadata": {
    "address": {
      "number": 21,
      "street": "Acacia Avenue"
    }
  },
  "registration": {
    "auth": {
      "id": "joebloggs@uk.ibm.com",
      "type": "person"
    },
    "date": "2014-08-21T18:25:43-05:00"
  }
}
```

Common Response Codes

- 200 (Success)

3.1.9 Delete a Registered Device

Unregister a device from an organization.

Request

DELETE **org_id**.internetofthings.ibmcloud.com/api/v0001/devices/**device_type/device_id**

Request Body

None

Response Body

None

Common Response Codes

- 204 (No Content) - The device was successfully deleted

3.2 Historical Event API

3.2.1 Authentication

The API is only accessible over HTTPS and is protected by HTTP Basic Authentication. You must provide a username and password with every request.

Username

A valid API key registered to the organization you want to work with.

Password

The authentication token corresponding to your API key.

3.2.2 View all events

View events across all devices registered to the organization

Request

GET **org_id**.internetofthings.ibmcloud.com/api/v0001/historian/

Request Headers

- cursorId
  - Use the cursorId response header returned by the initial request to iterate through the list of historical records
- Cookie
  - Required when setting the cursorId header. Use the iotHistorianSessionId cookie returned by the initial request

Request Body

None

Query Parameters

- evt_type - String
  - Restrict results only to those events published under this event identifier
- start - Number of milliseconds since January 1, 1970, 00:00:00 GMT)
  - Restrict results to events published after this date
- end - Number of milliseconds since January 1, 1970, 00:00:00 GMT)
  - Restrict results to events published before this date

Response Headers

- cursorId: use the cursorId response header returned by the initial request to iterate through the list of historical records
- Set-Cookie: iotHistorianSessionId

Response Body

Returns a list of historical data records sorted by timestamp in descending order

- device_id (String)
- device_type (String)
- evt_type (String)
- timestamp (Date)
- evt (Complex object)

Example Response

```
[
    {
        "device_id":"d:ey67sp:my-device-type:device0001",
        "device_type":"my-device-type",
        "evt_type":"sensor-event",
        "timestamp":{"$date":1407103315000},
        "evt":{"sensor1":76,"sensor2":0,"sensor3":79.98}
    },
    {
        "device_id":"d:ey67sp:my-device-type:device0002",
        "device_type":"my-device-type",
        "evt_type":"sensor-event",
        "timestamp":{"$date":1407103315000},
        "evt":{"sensor1":76,"sensor2":0,"sensor3":79.98}
    }
]
```


Common Response Codes

- 200 (success)

3.2.3 View events for a device type

View events across all devices of a specific type

Request

GET **org_id**.internetofthings.ibmcloud.com/api/v0001/historian/**device_type**

Request Headers

- cursorId
  - Use the cursorId response header returned by the initial request to iterate through the list of historical records
- Cookie
  - Required when setting the cursorId header. Use the iotHistorianSessionId cookie returned by the initial request

Request Body

None

Query Parameters

- evt_type - String
  - Restrict results only to those events published under this event identifier

- start - Number of milliseconds since January 1, 1970, 00:00:00 GMT)
  - Restrict results to events published after this date
- end - Number of milliseconds since January 1, 1970, 00:00:00 GMT)
  - Restrict results to events published before this date
- top - Number between 1 and 100
  - Restrict the number of records returned (default=100)
- summarize - Array
  - A list of fields from the JSON event payload on which to perform the aggregate function specified by the summarize_type parameter. The format for the parameter is {field1,field2,...,fieldN}
- summarize_type - String
  - The aggregation to perform on the fields specified by the summarize parameter:
    - avg (default)
    - count
    - min
    - max
    - sum
    - range
    - stdev
    - variance

Response Headers

- cursorId: use the cursorId response header returned by the initial request to iterate through the list of historical records
- Set-Cookie: iotHistorianSessionId

Response Body

Returns a list of historical data records sorted by timestamp in descending order

- device_id (String)
- evt_type (String)
- timestamp (Date)
- evt (Complex object)

Example Response

```
[
    {
        "device_id":"d:ey67sp:my-device-type:device0001",
        "evt_type":"sensor-event",
        "timestamp":{"$date":1407103315000},
```

```
        "evt":{"sensor1":76,"sensor2":0,"sensor3":79.98}
    },
    {
        "device_id":"d:ey67sp:my-device-type:device0002",
        "evt_type":"sensor-event",
        "timestamp":{"$date":1407103315000},
        "evt":{"sensor1":76,"sensor2":0,"sensor3":79.98}
    }
]
```

Common Response Codes

- 200 (success)

3.2.4 View events for a device

View events for a specific device

Request

GET **org_id**.internetofthings.ibmcloud.com/api/v0001/historian/**device_type/device_id**

Request Headers

- cursorId
  - Use the cursorId response header returned by the initial request to iterate through the list of historical records
- Cookie
  - Required when setting the cursorId header. Use the iotHistorianSessionId cookie returned by the initial request

Request Body

None

Query Parameters

- evt_type - String
  - Restrict results only to those events published under this event identifier
- start - Number of milliseconds since January 1, 1970, 00:00:00 GMT)
  - Restrict results to events published after this date
- end - Number of milliseconds since January 1, 1970, 00:00:00 GMT)
  - Restrict results to events published before this date

- top - Number between 1 and 100
  - Restrict the number of records returned (default=100).
- summarize - Array
  - A list of fields from the JSON event payload on which to perform the aggregate function specified by the summarize_type parameter. The format for the parameter is {field1,field2,...,fieldN}
- summarize_type - String
  - The aggregation to perform on the fields specified by the summarize parameter:
    - avg (default)
    - count
    - min
    - max
    - sum
    - range
    - stdev
    - variance

Response Headers

- cursorId: use the cursorId response header returned by the initial request to iterate through the list of historical records
- Set-Cookie: iotHistorianSessionId

Response Body

Returns a list of historical data records sorted by timestamp in descending order

- evt_type (String)
- timestamp (Date)
- evt (Complex object)

Example Response

```
[
    {
        "evt_type":"sensor-event",
        "timestamp":{"$date":1407103315000},
        "evt":{"sensor1":76,"sensor2":0,"sensor3":79.98}
    },
    {
        "evt_type":"sensor-event",
        "timestamp":{"$date":1407103315000},
        "evt":{"sensor1":76,"sensor2":0,"sensor3":79.98}
    }
]
```

Common Response Codes

- 200 (success)

CHAPTER 4

Client Libraries

4.1 Python Client Library - Introduction

Python 3.4 module for interacting with the IBM Internet of Things Foundation.

4.1.1 Dependencies

- paho-mqtt
- iso8601
- pytz
- requests

4.1.2 Installation

Install the latest version of the library with pip

```
[root@localhost ~]# pip install ibmiotf
```

4.1.3 Uninstall

Uninstalling the module is simple.

```
[root@localhost ~]# pip uninstall ibmiotf
```

4.2 Python Client Library - Devices

4.2.1 Constructor

The Client constructor accepts an options dict containing:

- org - Your organization ID
- type - The type of your device
- id - The ID of your device
- auth-method - Method of authentication (the only value currently supported is "token")
- auth-token - API key token (required if auth-method is "token")

```
import ibmiotf.device
try:
  options = {
    "org": organization,
    "type": deviceType,
    "id": deviceId,
    "auth-method": authMethod,
    "auth-token": authToken
  }
  client = ibmiotf.device.Client(options)
except ibmiotf.ConnectionException  as e:
  ...
```

Using a configuration file

```
import ibmiotf.device
try:
  options = ibmiotf.device.ParseConfigFile(configFilePath)
  client = ibmiotf.device.Client(options)
except ibmiotf.ConnectionException  as e:
  ...
```

The device configuration file must be in the following format:

```
[device]
org=$orgId
typ=$myDeviceType
id=$myDeviceId
auth-method=token
auth-token=$token
```

4.2.2 Handling commands

When the device client connects it automatically subscribes to any command for this device. To process specific commands you need to register a command callback method. The messages are returned as an instance of the Command class which has the following properties:

- payload - string
- format - string

- data - dict
- timestamp - datetime

```
def myCommandCallback(cmd):
  print("Command received: %s" % cmd.payload)
  if cmd.command == "setInterval":
    if 'interval' not in cmd.data:
      print("Error - command is missing required information: 'interval'")
    else:
      interval = cmd.data['interval']
  elif cmd.command == "print":
    if 'message' not in cmd.data:
      print("Error - command is missing required information: 'message'")
    else:
      print(cmd.data['message'])

...
client.connect()
client.commandCallback = myCommandCallback
```

4.2.3 Publishing events

Events can be published at any of the three *quality of service levels* defined by the MQTT protocol. By default events will be published as qos level 0.

Publish event using default quality of service

```
client.connect()
myData={'name' : 'foo', 'cpu' : 60, 'mem' : 50}
client.publishEvent("status", myData)
```

Publish event using user-defined quality of service

```
client.connect()
myQosLevel=2
myData={'name' : 'foo', 'cpu' : 60, 'mem' : 50}
client.publishEvent("status", myData, myQosLevel)
```

4.3 Python Client Library - Applications

4.3.1 Constructor

The Client constructor accepts an options dict containing:

- org - Your organization ID
- id - The unique ID of your application within your organization
- auth-method - Method of authentication (the only value currently supported is “apikey”)
- auth-key - API key (required if auth-method is “apikey”)

- auth-token - API key token (required if auth-method is "apikey")

```
import ibmiotf.application
try:
  options = {
    "org": organization,
    "id": appId,
    "auth-method": authMethod,
    "auth-key": authKey,
    "auth-token": authToken
  }
  client = ibmiotf.application.Client(options)
except ibmiotf.ConnectionException  as e:
  ...
```

Using a configuration file

```
import ibmiotf.application
try:
  options = ibmiotf.application.ParseConfigFile(configFilePath)
  client = ibmiotf.application.Client(options)
except ibmiotf.ConnectionException  as e:
  ...
```

The application configuration file must be in the following format:

```
[application]
org=$orgId
id=$myApplication
auth-method=apikey
auth-key=$key
auth-token=$token
```

4.3.2 Subscribing to device events

By default, this will subscribe to all events from all connected devices. Use the type, id and event parameters to control the scope of the subscription. A single client can support multiple subscriptions.

Subscribe to all events from all devices

```
client.connect()
client.subscribeToDeviceEvents()
```

Subscribe to all events from all devices of a specific type

```
client.connect()
client.subscribeToDeviceEvents(deviceType=myDeviceType)
```

Subscribe to a specific event from all devices

```
client.connect()
client.subscribeToDeviceEvents(event=myEvent)
```

Subscribe to a specific event from two different devices

```
client.connect()
client.subscribeToDeviceEvents(deviceType=myDeviceType, deviceId=myDeviceId, event=myEvent)
client.subscribeToDeviceEvents(deviceType=myOtherDeviceType, deviceId=myOtherDeviceId, event=myEvent)
```

4.3.3 Handling events from devices

To process the events received by your subscroptions you need to register an event callback method. The messages are returned as an instance of the Event class:

- event.device - string (uniquely identifies the device across all types of devices in the organization
- event.deviceType - string
- event.deviceId - string
- event.event - string
- event.format - string
- event.data - dict
- event.timestamp - datetime

```
def myEventCallback(event):
  str = "%s event '%s' received from device [%s]: %s"
  print(str % (event.format, event.event, event.device, json.dumps(event.data)))

...
client.connect()
client.deviceEventCallback = myEventCallback
client.subscribeToDeviceEvents()
```

4.3.4 Subscribing to device status

By default, this will subscribe to status updates for all connected devices. Use the type and id parameters to control the scope of the subscription. A single client can support multiple subscriptions.

Subscribe to status updates for all devices

```
client.connect()
client.subscribeToDeviceStatus()
```

Subscribe to status updates for all devices of a specific type

```
client.connect()
client.subscribeToDeviceStatus(deviceType=myDeviceType)
```

Subscribe to status updates for two different devices

```
client.connect()
client.subscribeToDeviceStatus(deviceType=myDeviceType, deviceId=myDeviceId)
client.subscribeToDeviceStatus(deviceType=myOtherDeviceType, deviceId=myOtherDeviceId)
```

4.3.5 Handling status updates from devices

To process the status updates received by your subscriptions you need to register an event callback method. The messages are returned as an instance of the Status class:

The following properties are set for both "Connect" and "Disconnect" status events:

- status.clientAddr - string
- status.protocol - string
- status.clientId - string
- status.user - string
- status.time - datetime
- status.action - string
- status.connectTime - datetime
- status.port - int

The following properties are only set when the action is "Disconnect":

- status.writeMsg - int
- status.readMsg - int
- status.reason - string
- status.readBytes - int
- status.writeBytes - int

```
def myStatusCallback(status):
  if status.action == "Disconnect":
    str = "%s - device %s - %s (%s)"
    print(str % (status.time.isoformat(), status.device, status.action, status.reason))
  else:
    print("%s - %s - %s" % (status.time.isoformat(), status.device, status.action))

...
client.connect()
client.deviceStatusCallback = myStatusCallback
client.subscribeToDeviceStstus()
```

4.3.6 Publishing events from devices

Applications can publish events as if they originated from a Device

```
client.connect()
myData={'name' : 'foo', 'cpu' : 60, 'mem' : 50}
client.publishEvent(myDeviceType, myDeviceId, "status", myData)
```

4.3.7 Publishing commands to devices

Applications can publish commands to connected devices

```
client.connect()
commandData={'rebootDelay' : 50}
client.publishCommand(myDeviceType, myDeviceId, "reboot", myData)
```

4.3.8 Retrieve device details

Retrieve details of all registered devices

```
deviceList = client.api.getDevices()
print(deviceList)
```

Retrieve details of a specific device

```
device = client.api.getDevice(deviceType, deviceId)
print(device)
```

4.3.9 Register a new device

```
device = client.api.registerDevice(deviceType, deviceId, metadata)
print(device)
print("Generated Authentication Token = %s" % (device['password']))
```

4.3.10 Delete a device

```
try:
  client.api.deleteDevice(deviceType, deviceId)
except Exception as e:
  print(str(e))
```

4.3.11 Access historical event data

Get historical event data for a specific device

```
result = client.api.getHistoricalEvents(deviceType, deviceId)
print(result)
```

Get historical event data for all devices of a specific type

```
result = client.api.getHistoricalEvents(deviceType)
print(result)
```

Get historical event data for all devices of all types

```
result = client.api.getHistoricalEvents()
print(result)
```

CHAPTER 5

Messaging

5.1 MQTT

The primary mechanism that devices and applications use to communicate with the IBM Internet of Things Foundation is MQTT; this is a protocol designed for the efficient exchange of real-time data with sensor and mobile devices.

MQTT runs over TCP/IP and, while it is possible to code directly to TCP/IP, you might prefer to use a library that handles the details of the MQTT protocol for you. You will find there's a wide range of MQTT client libraries available at mqtt.org, with the best place to start looking being the Eclipse Paho project. IBM contributes to the development and support of many of these libraries.

MQTT 3.1 is the version of the protocol that is in widest use today. Version 3.1.1 contains a number of minor enhancements, and has been ratified as an OASIS Standard.

One reason for using version 3.1.1 is that the maximum length of the MQTT Client Identifier (ClientId) is increased from the 23 character limit imposed by 3.1. The IoT service will often require longer ClientId's and will accept long ClientId's with either version of the protocol however some 3.1 client libraries check the ClientId and enforce the 23 character limit.

5.1.1 MQTT client connection

Every registered organization has a unique endpoint which must be used when connecting MQTT clients for applications and devices in that organization.

org_id.messaging.internetofthings.ibmcloud.com

> **Warning:** Currently, your device will also be able to connect to **messaging.internetofthings.ibmcloud.com**, however we highly discourage users from writing client code using this domain name, as it **will** stop working at some point in the near future.

5.1.2 Unencrypted client connection

Connect on port **1883**

Important: All information your device submits is being sent in plain text (including the authentication credentials for your device).

5.1.3 Encrypted client connection

Connect on port **8883** or **443** for websockets.

In many client libraries you will need to provide the server's public certificate in pem format. The following file contains the entire certificate chain for *.messaging.internetofthings.ibmcloud.com: messaging.pem

Tip: Some SSL client libraries have been shown to not handle wildcarded domains, in which case, if you can not change libraries, you will need to turn off certificate checking.

Note: The IoT Foundation requires TLS v1.2. We suggest the following cipher suites: ECDHE-RSA-AES256-GCM-SHA384, AES256-GCM-SHA384, ECDHE-RSA-AES128-GCM-SHA256 or AES128-GCM-SHA256 *(as of Jun 1 2015)*.

5.1.4 Device and application clients

We define two primary classes of thing: Devices & Applications

The class of thing that your MQTT client identifies itself to the service as will determine the capabilities of your client once connected as well as the mechanism through which you will need to authenticate.

Applications and devices also work with different MQTT topic spaces. Devices work within a device-scoped topic space, whereas applications have full access to the topic space for an entire organization.

- MQTT Connectivity for Devices
- MQTT Connectivity for Applications

5.1.5 Quality of service

The MQTT protocol provides three qualities of service for delivering messages between clients and servers: "at most once", "at least once" and "exactly once". Events and commands can be sent using any quality of service level, however you should carefully consider whether what the right level is for your needs. It is not a simple case that QoS2 is "better" than QoS0.

At most once (QoS0)

The message is delivered at most once, or it might not be delivered at all. Delivery across the network is not acknowledged, and the message is not stored. The message could be lost if the client is disconnected, or if the server fails. QoS0 is the fastest mode of transfer. It is sometimes called "fire and forget".

The MQTT protocol does not require servers to forward publications at QoS0 to a client. If the client is disconnected at the time the server receives the publication, the publication might be discarded, depending on the server implementation.

Tip: When sending real-time data on an interval we recommend using QoS0. If a single message goes missing it

does not really matter as another message will be sent shortly after containing newer data. In this scenario the extra cost of using higher quality of service does not result in any tangible benefit.

At least once (QoS1)

The message is always delivered at least once. It might be delivered multiple times if there is a failure before an acknowledgment is received by the sender. The message must be stored locally at the sender, until the sender receives confirmation that the message has been published by the receiver. The message is stored in case the message must be sent again.

Exactly once (QoS2)

The message is always delivered exactly once. The message must be stored locally at the sender, until the sender receives confirmation that the message has been published by the receiver. The message is stored in case the message must be sent again. QoS2 is the safest, but slowest mode of transfer. A more sophisticated handshaking and acknowledgement sequence is used than for QoS1 to ensure no duplication of messages occurs.

Tip: When sending commands we recommend using QoS2. In many cases, when processing commands you want to know that the command will only be actioned, and that it will be actioned only once. This is one of the clearest examples of when the additional overhead of QoS2 has a clear benefit.

Subscription Buffers and Clean Session

Each subscription from either a device or application is allocated a buffer of 5000 messages. This allows for any application or device to fall behind the live data it is processing and build up a backlog of up to 5000 pending messages for each subscription it has made. Once the buffer fills up, any new message will result in the oldest message in the buffer being discarded.

The subscription buffer can be accessed using MQTT clean session option. If clean session is set to true, a subscriber will start receiving messages from the buffer. If it is false the buffer is reset.

Note: This limit applies regardless of the quality of service setting used. Thus it is possible that a message sent at QoS1 or QoS2 may not be delivered to an application that is unable to keep up with the messages rate for the subscription(s) it has made.

5.2 MQTT Connectivity for Devices

A Device must authenticate using a client ID in the following format:

d:**org_id**:**device_type**:**device_id**

- A Device must be registered before it can connect
- When connecting to the QuickStart service no authentication (or registration) is required

---

5.2.1 MQTT client identifier

- Supply a client id of the form **d**:**org_id**:**device_type**:**device_id**
- **d** idenfities your client as a device
- **org_id** is your unique organization ID, assigned when you sign up with the service. It will be a 6 character alphanumeric string.
- **device_type** is intended to be used as an identifer of the type of device connecting, it may be useful to think of this as analogous to a model number.
- **device_id** must uniquely identify a device across all devices of a specific device_type, it may be useful to think of this as analogous to a serial number.

5.2.2 MQTT authentication

Username

The service currently only supports token-based authentication for devices, as such there is only one valid username for devices today.

A value of **use-token-auth** indicates to the service that the authentication token for the device will be passed as the password for the MQTT connection.

Password

When using token based authentication submit the device authentication token as the password when making your MQTT connection.

5.2.3 Publishing events

- Publish to topic iot-2/evt/**event_id**/fmt/**format_string**

5.2.4 Subscribing to commands

- Subscribe to topic iot-2/cmd/**command_id**/fmt/**format_string**

5.2.5 QuickStart restrictions

If you are writing device code that wants to support use with QuickStart you must take into account the following features present in the registered service that are not supported in QuickStart:

- Subscribing to commands
- MQTT connection over SSL
- Clean or durable sessions

5.3 MQTT Connectivity for Applications

An Application must authenticate using a client ID in the following format:

a:**org_id**:**app_id**

- We do not impose any rules on the **app_id** component of the client ID
- When connecting to the QuickStart service no authentication is required
- An Application does not need to be registered before it can connect

---

5.3.1 MQTT client identifier

- Supply a client id of the form **a**:**org_id**:**app_id**
- **a** indicates the client is an application
- **org_id** is your unique organization ID, assigned when you sign up with the service. It will be a 6 character alphanumeric string.
- **app_id** is a user-defined unique string identifier for this client.

Note: Only one MQTT client can connect using any given client ID. As soon as a secnd client in your organization connects using an **app_id** that you have already connected the first client will be disconnected.

5.3.2 MQTT authentication

Applications require an API Key to connect into an Organization. When an API Key is registered a token will be generated that must be used with that API key.

The API key will look something like this: a-**org_id**-a84ps90Ajs

The token will look something like this: MP$08VKz!8rXwnR-Q*

When making an MQTT connection using an API key the following applies:

- MQTT client ID: a:**orgid**:**app_id**
- MQTT username must be the API key: a-**org_id**-a84ps90Ajs
- MQTT password must be the authentication token: MP$08VKz!8rXwnR-Q*

5.3.3 Publishing device events

An application can publish events as if they came from any registered device.

- Publish to topic iot-2/type/**device_type**/id/**device_id**/evt/**event_id**/fmt/**format_string**

Tip: You may have a number of devices that are already generating bespoke data that you wish to send to IOTF. One way to get that data into the service would be to write an application that processes the data and publishes it to IOTF.

5.3.4 Publishing device commands

An application can publish a command to any registered device.

- Publish to topic iot-2/type/**device_type**/id/**device_id**/cmd/**command_id**/fmt/**format_string**

5.3.5 Subscribing to device events

An application can subscribe to events from one or more devices.

- Subscribe to topic iot-2/type/**device_type**/id/**device_id**/evt/**event_id**/fmt/**format_string**

Note: The MQTT "any" wildcard character (+) may be used for any of the following components if you want to subscribe to more than one type of event, or events from more than a single device.

- device_type
- device_id
- event_id
- format_string

5.3.6 Subscribing to device commands

An application can subscribe to commands being sent to one or more devices.

- Subscribe to topic iot-2/type/**device_type**/id/**device_id**/cmd/**command_id**/fmt/**format_string**

Note: The MQTT "any" wildcard character (+) may be used for any of the following components if you want to subscribe to more than one type of event, or events from more than a single device.

- device_type
- device_id
- cmd_id
- format_string

5.3.7 Subscribing to device status messages

An application can subscribe to monitor status of one or more devices.

- Subscribe to topic iot-2/type/**device_type**/id/**device_id**/mon

---

Note: The MQTT "any" wildcard character (+) may be used for any of the following components if you want to subscribe to updates from more than one device.

- device_type
- device_id

5.3.8 Subscribing to application status messages

An application can subscribe to monitor status of one or more applications.

- Subscribe to topic iot-2/app/**app_id**/mon

Note: The MQTT "any" wildcard character (+) may be used for **app_id** if you want to subscribe for updates for all applications.

5.3.9 QuickStart restrictions

If you are writing application code that wants to support use with QuickStart you must take into account the following features present in the registered service that are not supported in QuickStart:

- Publishing commands
- Subscribing to commands
- Use of the MQTT "any" wildcard character (+) for the following topic components:
  - device_type
  - app_id
- MQTT connection over SSL

5.4 Message Payload

The Foundation supports sending and recieving messages in any format, however we recommend the use of JSON and the IOTF event format specification.

5.4.1 IOTF JSON Payload Specification

It is simple to create a JSON message that meets the IOTF specification.

- The message must be a valid JSON object (not an array) with only two top level elements: **d** and **ts**
- The message must be UTF-8 encoded

Data

The **d** element is where you include all data for the event (or command) being transmitted in the message.

- This element is required for your message to meet the IOTF message specification.
- This must always be a JSON object (not an array)
- In the case where you wish to send no data the **d** element should still be present, but contain an empty object.

Example 1 - Simple Data

```
{
  "d": {"msg": "Hello World"}
}
```

Example 2 - Complex data

```
{
  "d": {
    "host": "IBM700-R9E683D",
    "mem": 54.9,
    "network": {
      "up": 1.22,
      "down": 0.55
    },
    "cpu": 1.3,
  }
}
```

Example 3 - No data

```
{
  "d": {}
}
```

Timestamp

The **ts** element allows you to associate a timestamp with the event (or command). This is an optional element, if included its value should be a valid ISO8601 encoded timestamp string.

```
{
  "d": {
    "host": "IBM700-R9E683D",
    "mem": 54.9,
    "network": {
      "up": 1.22,
      "down": 0.55
    },
    "cpu": 1.3,
  },
  "ts": "2014-12-30T14:47:36+00:00"
}
```

5.4.2 Custom JSON payloads

The IOT Foundation is designed to be open, you may send your event and command data in any format you choose, however if you choose to send data in a custom format it will limit some features of the service which can only function with a known payload format.

Below are a number of example payloads that are close the to the IOTF specification, but do not quite match it. Each would be treated as a custom JSON payload.

Example 1

Root node is a JSON array

```
[
  {
    "d": {
      "myName": "Stuart's Pi",
      "cputemp": 46,
      "sine": -10,
      "cpuload": 1.45
    }
  },
  {
    "d": {
      "myName": "Stuart's Pi",
      "cputemp": 46,
      "sine": -10,
      "cpuload": 1.45
    }
  }
]
```

Example 2

"d" node is a JSON array

```
{
  "d": ["green", "yellow"]
}
```

Example 3

Unexpected node at root level

```
{
  "d": {},
  "temp": 60,
  "ts": "2014-12-30T14:47:36+00:00"
}
```

CHAPTER 6

Contribute

If it's not working for you, it's not working for us. The source of this documentation is available on GitHub, we welcome both suggestions for improvement and community contributions via the use of issues and pull requests against our repository.