IBM IOT Foundation Documentation

Release 1.0

David Parker

March 11, 2016

1 Concepts 3

2 Feature Overview 7

3 Quickstart 9

4 Register 13

5 HTTP API for Applications 23

6 MQTT Connectivity for Applications 25

7 Python for Application Developers 31

8 Node.js for Application Developers 47

9 Application 49

10 Java for Application Developers 57

11 C# for Application Developers 65

12 HTTP API for Devices 69

13 MQTT Connectivity for Devices 71

14 Device Management Protocol 75

15 Device Management Requests 87

16 Python for Device Developers 97

17 Node.js for Device Developers 101

18 Java for Device Developers 105

19 C# for Device Developers 111

20 Embedded C for Device Developers 113

21 mBed C++ for Device Developers 117

22 MQTT Connectivity for Gateways 123

23 MQTT 127

24 Device Model 131

25 External Service Integrations 137

26 Securing the Watson IoT platform 139

27 Contribute 143

Welcome to the documentation/reference material for IBM's IoT Platform service. If you are looking for tutorials you can find them on developerWorks Recipes.

CHAPTER 1

Concepts

1.1 Organizations

When you register with the Watson IoT platform 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 Watson IoT platform 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.

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 Watson IoT platform with an authentication token that will only be accepted for that device.
- Devices must be registered before they can connect to the Watson IoT platform.

1.2.1 Managed and Unmanaged Devices

Managed devices are defined as devices which contain a management agent. A management agent is a set of logic which allows the device to interact with the Watson IoT platform Device Management service via the Device Management protocol. Managed devices can perform device management operations including location updates, firmware download and updates, and reboot and factory reset.

An **unmanaged device** is a device without a management agent. An unmanaged device can still connect to the Watson IoT platform and send and receive events and commands. However, it cannot send any device management requests, or perform any of the device management operations.

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 Watson IoT platform with an API key and a unique application ID.
- Applications do not need to be registered before they can connect to the Watson IoT platform, however they must present a valid API key that has previously been registered.

1.4 Gateway Devices

Important: Gateway support is currently available as part of a limited beta. Future updates may include changes incompatible with the current version of this feature. Try it out and let us know what you think

- Gateways are a specialized class of Device. They have the combined capabilities of an Application and a Device allowing them to serve as access points providing connectivity to the service for other devices without the ability to directly connect.
- Gateway devices can register new devices and can send and receive data on behalf of devices connected to them.
- Gateway Devices must be registered before they can connect to the service

1.5 Events

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

When an event is received by the Watson IoT platform from a device 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.

1.5.1 Historical Event Storage

Historical event storage allows users to store the data from devices added to their Watson IoT platform organization. Historical event storage activity and duration can be controlled from the settings panel in the Watson IoT platform dashboard.

Tip: When changing settings for historical data storage, keep in mind that storing data for longer periods of time will affect your billing. Also, care should be taken when disabling historical data storage, as upon disabling data storage for your organization, all stored data will be deleted, and cannot be recovered.

Warning: Historical event storage uses an MQTT Quality of Service level of 0 (delivery at most once), so some data may be lost.

1.6 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.

CHAPTER 2

Feature Overview

2.1 Device Registry

Manage your inventory, configure security, and store metadata for millions of unique devices. Define device types to represent individual device models and apply default metadata to all devices of that type.

2.2 Connectivity

Securely connect your devices, gateways and applications directly to the Watson IoT platform via MQTT. See the section on MQTT in the *reference material* to learn more about the advantages of using this protocol. Model the data from your device as events and control the flow of events into your applications.

2.3 Gateway Support

In many cases a direct connection can not be made between the service and a device, the Watson IoT platform allows gateway devices to connect that can provide indirect connectivity for multiple devices.

Important: This feature is currently available as part of a limited beta. Future updates may include changes incompatible with the current version of this feature. Try it out and let us know what you think

2.4 Device Management

Optionally, allow the Watson IoT platform to manage the lifecycle of your devices by implementing support for the Watson IoT platform's device management protocol in your devices. The means by which the device connects to the service does not affect the device management protocol, which functions the same for directly connected, indirectly connected, and gateway devices.

2.5 External Service Integration

The Watson IoT platform supports integration with external services to bring data and operations supported by other online services into the platform, allowing your application and device developers to seemlessly interact with those

services without ever leaving the comfort of the Watson IoT platform APIs.

Important: This feature is currently available as part of a limited beta. Future updates may include changes incompatible with the current version of this feature. Try it out and let us know what you think

2.6 Historian

Configure the Watson IoT platform to store a record of the events your devices generate.

Warning: Message format restrictions apply to the Historian
The historian feature only supports JSON messages meeting specific critera:

- 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 Watson IoT platform 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.

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 timestamp.
Example

```
{
  "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"
}
```

CHAPTER 3

Quickstart

Quickstart is an open sandbox allowing developers to quickly and easily get devices connected to the Watson IoT platform with registration required. Any device that can run an MQTT client can be connected to Quickstart within mimimum fuss, DeveloperWorks Recipes features dozens of community produced tutorials for connecting different devices to the service, including but not limited to:

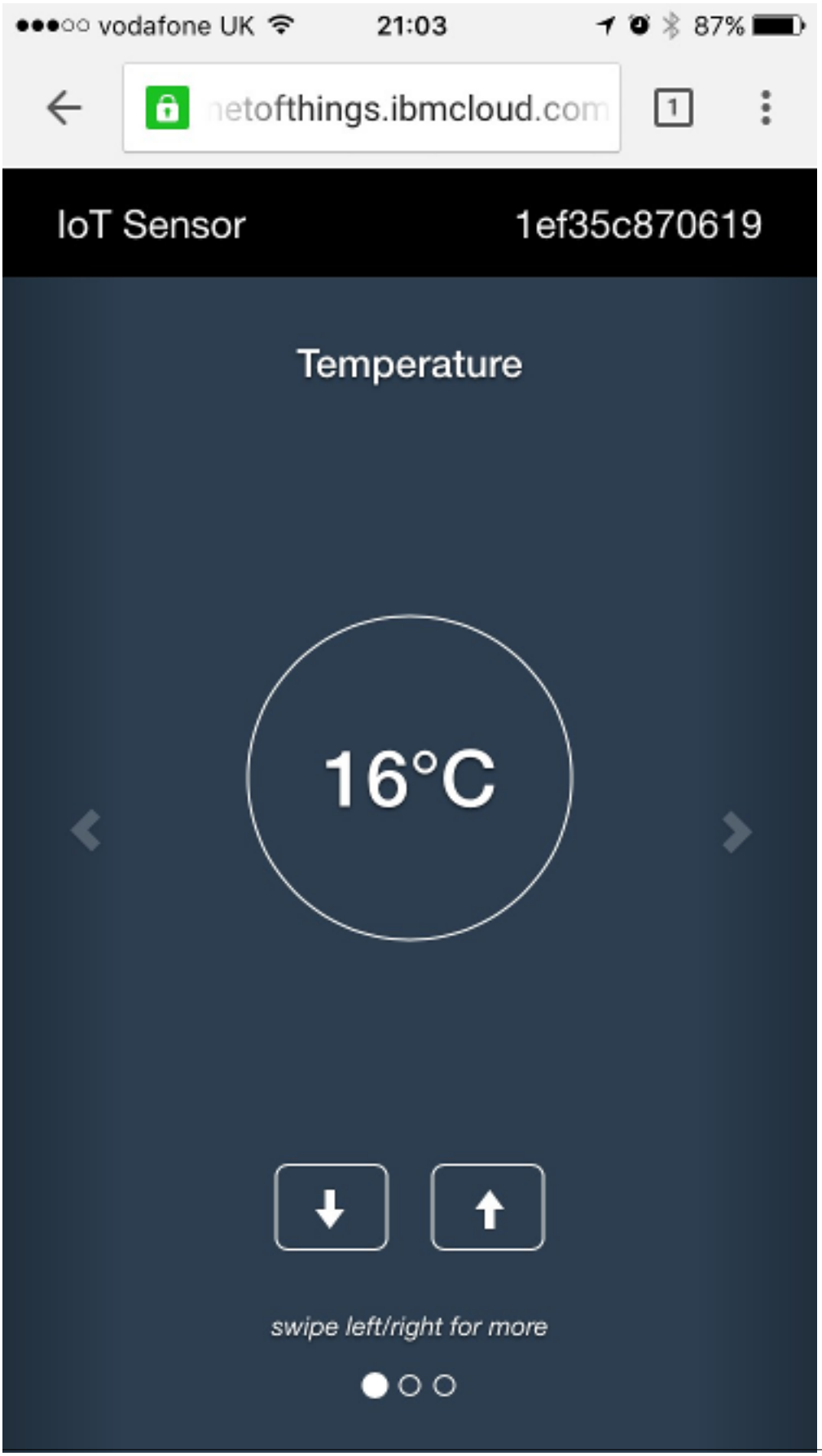
- OpenBlocks IoT BX1G
- Reactive Blocks

3.1 Simulated Device

In addition to these, we have developed a simple browser-based simulated device, designed for mobile devices, that can be used to connect any device with a web browser to the service. Open the following URL on a mobile phone or tablet in your favourite browser: http://quickstart.internetofthings.ibmcloud.com/iotsensor.

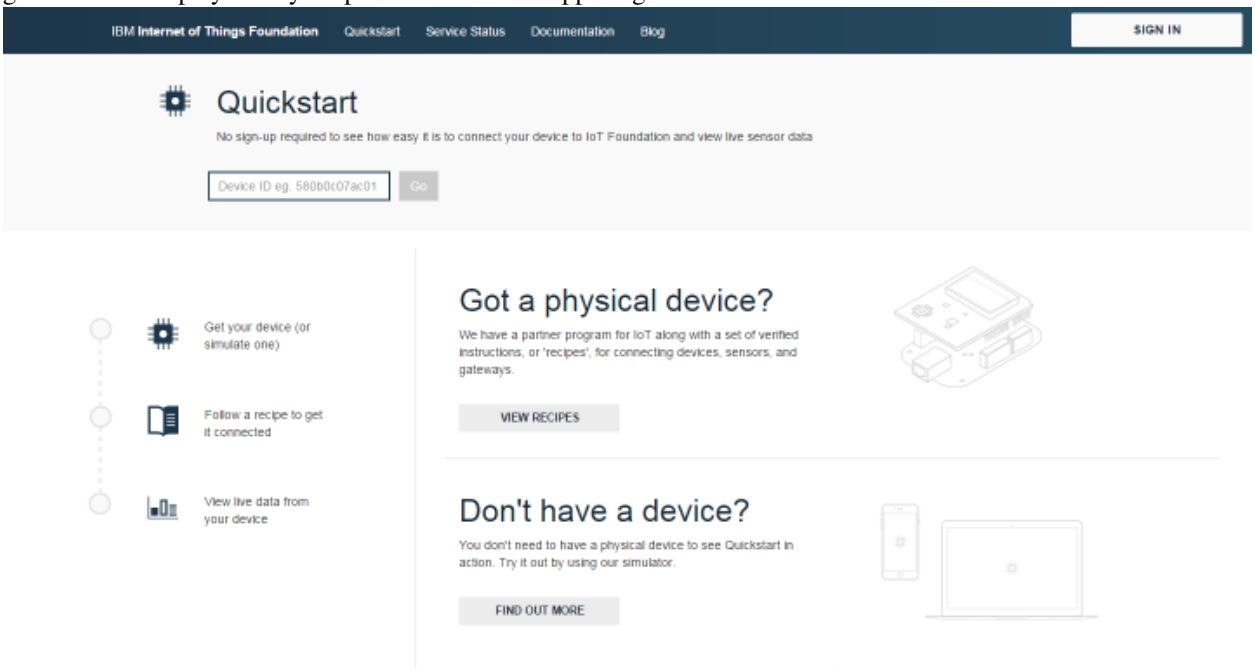
This will launch a browser-based simulated device connecting to the Watson IoT platform. There are three sensors which you can manipulate using the on-screen controls:

- Temperature
- Humidity
- Object temperature

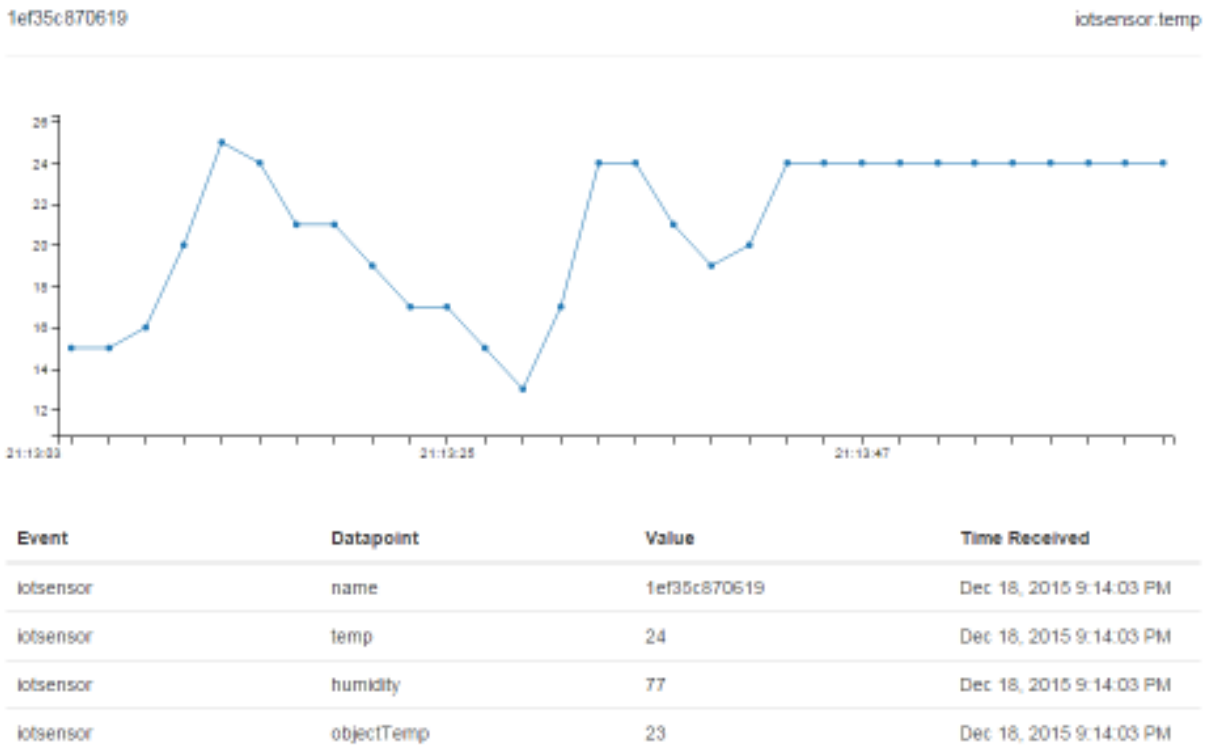

vodafone UK
21:03
87%
etofthings.ibmcloud.com
1
IoT Sensor
1ef35c870619
Temperature
16°C
swipe left/right for more

3.2 Data Visualization

Leaving the simulated device running on your phone/tablet launch the Quickstart application and enter the 12 character generated ID displayed on your phone/tablet in the upper right-hand corner.



Now, as you adjust the sensor values in your simulated device you will be able to see the data from your device visualized in real time in the Quickstart application.



Event	Datapoint	Value	Time Received
iotsensor	name	1ef35c870619	Dec 18, 2015 9:14:03 PM
iotsensor	temp	24	Dec 18, 2015 9:14:03 PM
iotsensor	humidity	77	Dec 18, 2015 9:14:03 PM
iotsensor	objectTemp	23	Dec 18, 2015 9:14:03 PM

3.3 Mosquitto Demonstration

Mosquitto is a cross platform open source MQTT client that is perfect for experimenting with the Watson IoT platform service. Once you have installed mosquitto pick an `applicationId`, a `deviceId` that have a good chance to be unique (otherwise your experiment will clash with someone else running through this same demonstration).

Note: There are a couple of restrictions you must consider when dedicing on these IDs:

- Maximum length of 36 characters
- Must comprise only alpha-numeric characters (`a-z`, `A-Z`, `0-9`) and the following special characters:
  - dash (`-`)
  - underscore (_)
  - dot (`.`)

With your `applicationId` and `deviceId` in hand the first thing you want to do is create a connection representing your application using `mosquitto_sub`:

```
[user@host ~]$ mosquitto_sub -h quickstart.messaging.internetofthings.ibmcloud.com -p 1883 -i "a:quic
```

Leave that process running, it is now time to turn your attention towards creating your device. We will connect a device of type `mosquitto` and send two events to the service using `mosquitto_pub`:

```
[user@host ~]$ mosquitto_pub -h quickstart.messaging.internetofthings.ibmcloud.com -p 1883 -i "d:quic
[user@host ~]$ mosquitto_pub -h quickstart.messaging.internetofthings.ibmcloud.com -p 1883 -i "d:quic
```

Returning to your application terminal you should see the two events that you published almost instantly:

```
[user@host ~]$ mosquitto_sub -h quickstart.messaging.internetofthings.ibmcloud.com -p 1883 -i "a:quic
{"helloworld": 1}
{"helloworld": 2}
```

That's all there is to it. You have successfully connected a device and an application to the Watson IoT platform over MQTT, sent an event from the device to the service and recieved that event in your application.

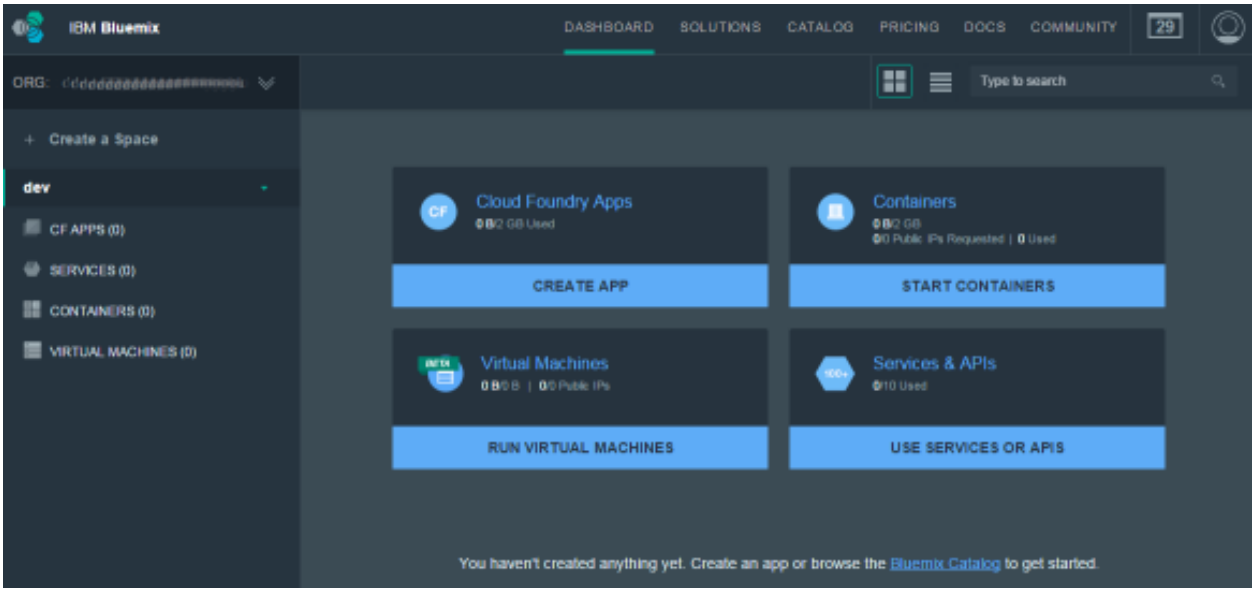
CHAPTER 4

Register

Registering for an account on the Watson IoT platform and connecting your first device is simple and will take less than 10 minutes to complete.

4.1 Create a Bluemix Account

First you will need to register for a Bluemix account. Once you have verified your e-mail address you should find yourself looking at an empty dashboard:



4.2 Choose your Region

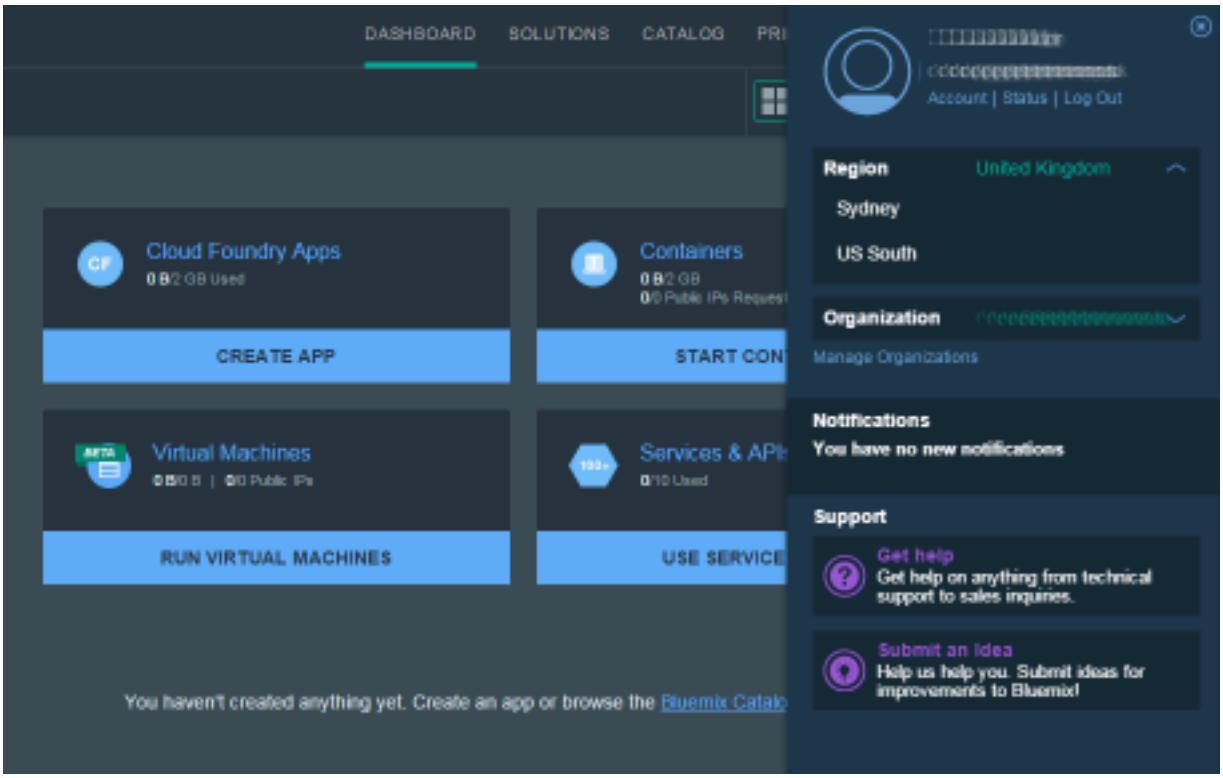
Now, before you go any further you need to consider data soveriegnty and make a decision as to which of the available regions you want to register your service instance with. Currently the Watson IoT platform is available in two regions:

- United Kingom
- United States

Note: Data sovereignty is the concept that information which has been converted and stored in binary digital form is subject to the laws of the country in which it is located.

The region your Watson IoT platform instance will be provisioned in will be determined by the Bluemix region you are using. You can switch Bluemix regions via the controls in the top right corner of the Bluemix dashboard. You can also determine which region are currently have selected by the URL in your browser:

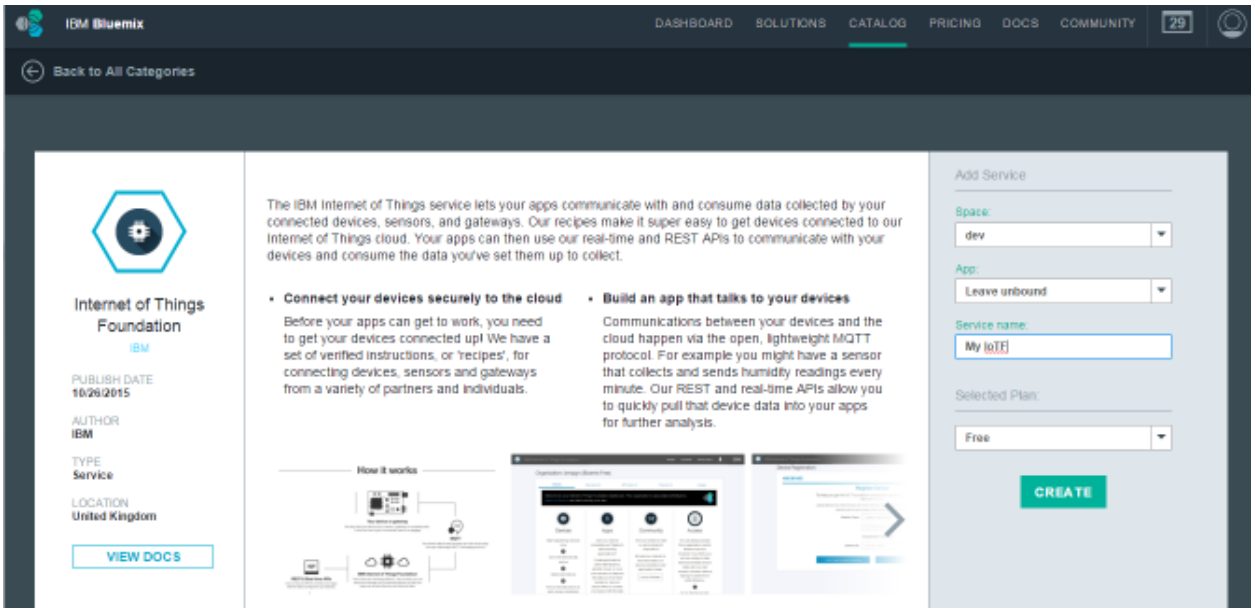
- United Kingdom: console.**eu-gb**.bluemix.net
- United States: console.**ng**.bluemix.net



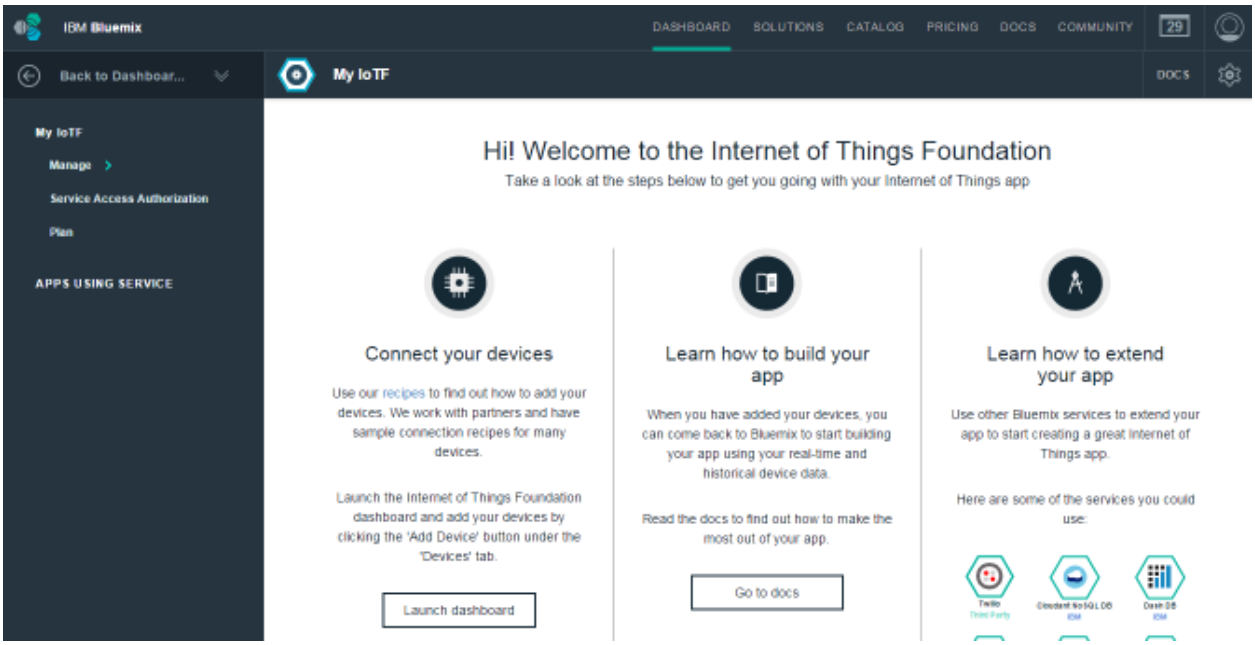
4.3 Provision a Watson IoT platform Service Instance

From anywhere in the Bluemix dashboard click **catalog** in the top navigation, this will take you to a list of all services available on Bluemix.

1. Type `Internet of Things` into the search box at the top of the page to filter out all the other services.
2. Click `View More` on the **Watson IoT platform** service tile.
3. Optionally give your service instance a name (or simply accept the generated name)
4. Click `Create`



After a short delay you will find yourself the proud owner of an organization on the IBM Watson IoT platform.



4.4 Additional Setup and Experimentation

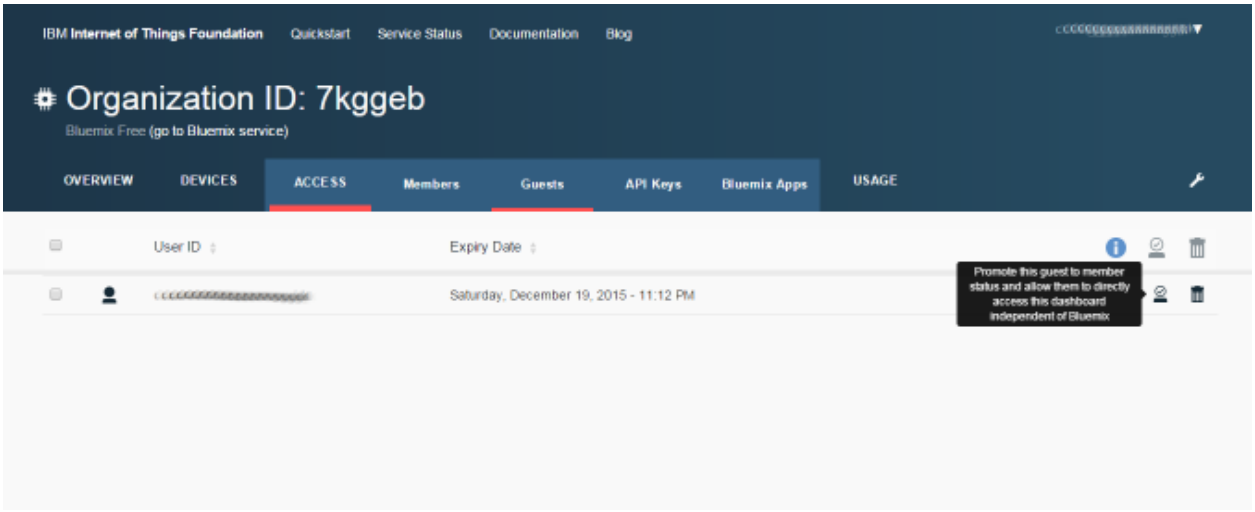
Click the `Launch Dashboard` button when viewing your new service instance within Bluemix, this will take you to your Watson IoT platform Dashboard.

4.4.1 Configure Access

You are currently accessing the dashboard through a 24 hour access token generated by Bluemix when you clicked `Launch Dashboard`, when this expires you will need to return to Bluemix and relaunch the dashboard. I think we

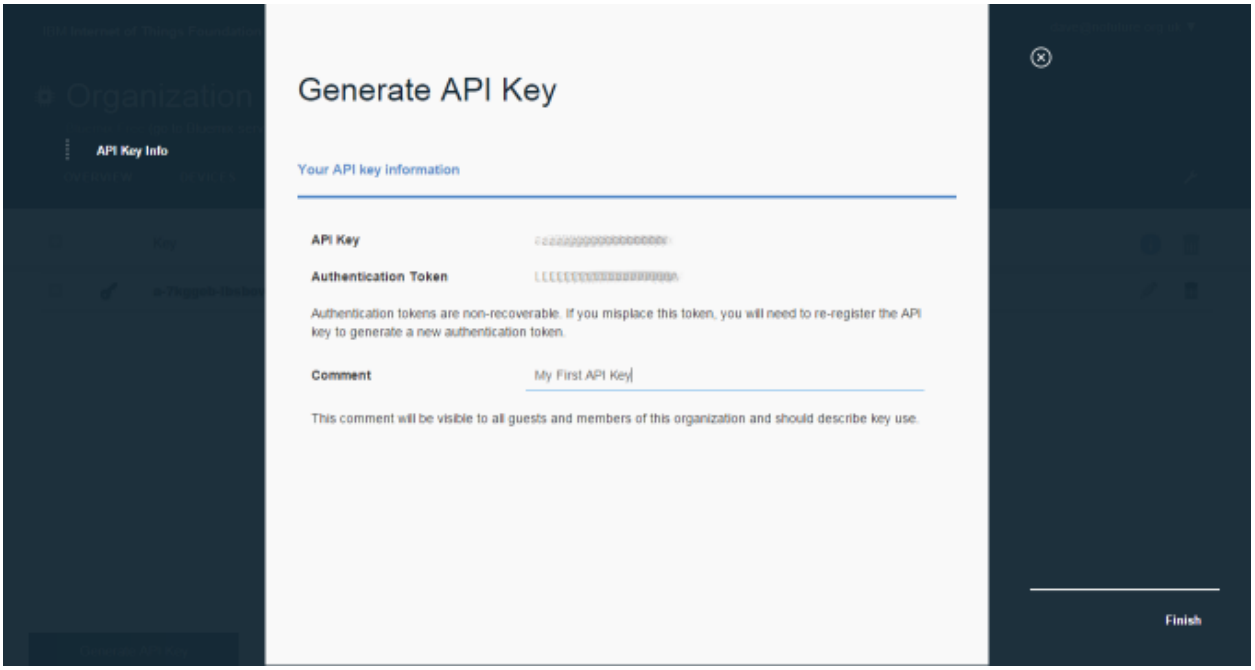
can all agree it's preferable not to have to do this, so let's promote you to a permanent member of the organization.

In the navigation menu select `Access` and then `Guests`, you should see the e-mail address you signed up to Bluemix with in there and the expiration date for your session. Click the `Promote` button to set yourself up as a permanent member of the organization, your ID will disappear from the list, and can now be found under `Access > Members`.



4.4.2 Connect an Application

Applications require an `API key` to connect to the service. Let's set one up. Navigate to `Access > API Keys` and click `Generate API Key`. Take a note of your `API Key` and `Authentication Token` - you will need them shortly.



Now we have an API key we want to run an application, the simpler the better.

Note: The Watson IoT platform has client library support for multiple languages, but for the purpose of this getting

started tutorial we're going to stick with Python. If you don't already have Python installed now is the time to do so.

Install the latest version of the Watson IoT platform python client library using pip

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

Now, create a really simple application that will connect using the API key you just created:

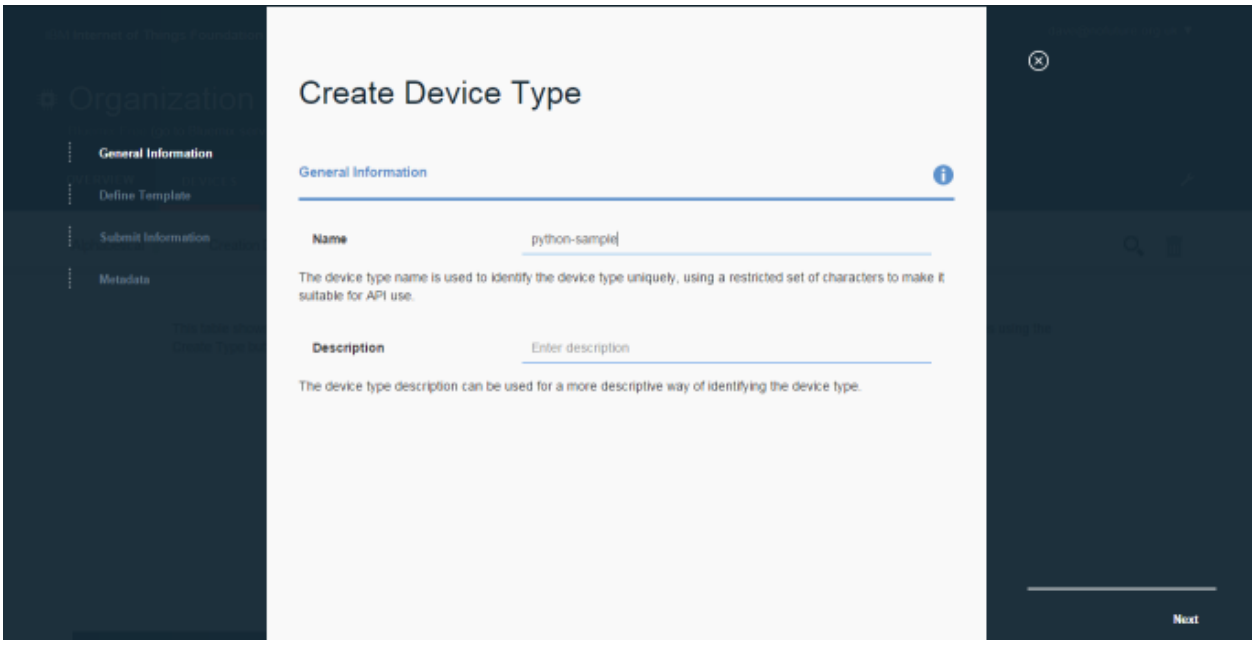
```
import signal
import time
import sys
import json
import ibmiotf.application

def myEventCallback(myEvent):
  print("%-33s%-32s%s: %s" % (myEvent.timestamp.isoformat(), myEvent.device, myEvent.event, json.dump

def interruptHandler(signal, frame):
  client.disconnect()
  sys.exit(0)


options = {
  "org": "<INSERT_ORGANIZATION_ID>",
  "id": "MyFirstApplication",
  "auth-method": "apikey",
  "auth-key": "<INSERT_API_KEY>",
  "auth-token": "<INSERT_AUTH_TOKEN>"
}

try:
  client = ibmiotf.application.Client(options)
  client.connect()
except Exception as e:
  print(str(e))
  sys.exit()

print("(Press Ctrl+C to disconnect)")
client.deviceEventCallback = myEventCallback
client.subscribeToDeviceEvents()

while True:
  time.sleep(1)
```

When you launch the application you will see nothing overly exciting

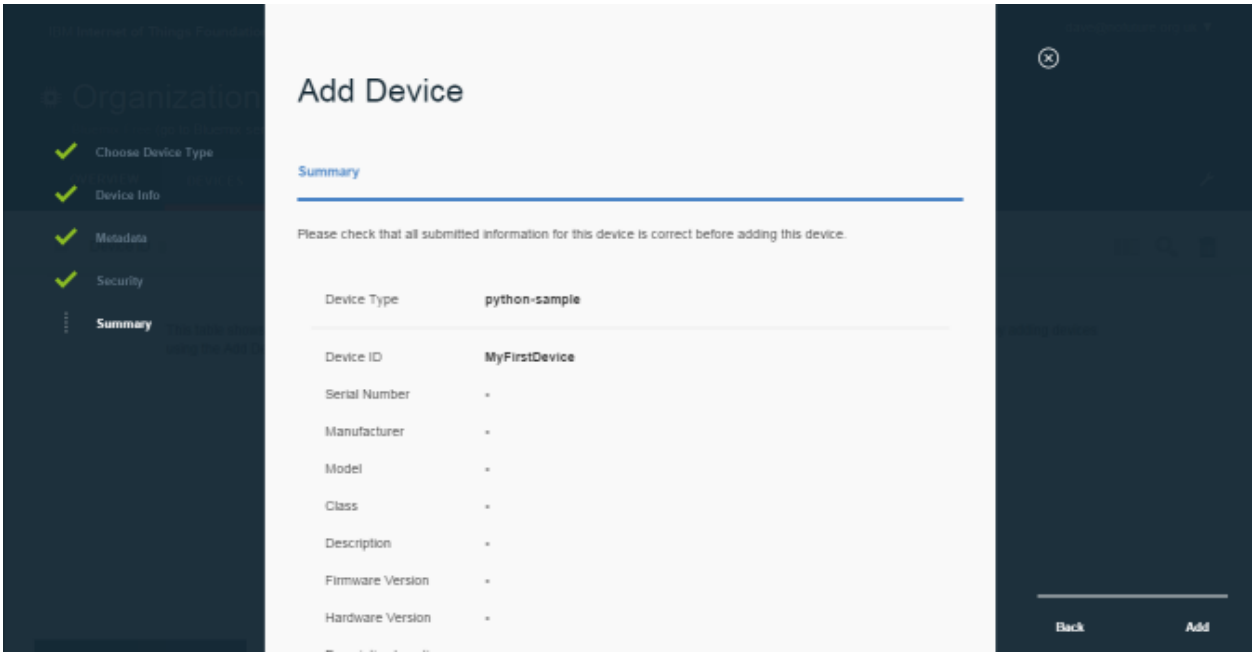
```
[user@host ~]$ python test.py
(Press Ctrl+C to disconnect)
2015-12-19 00:04:28,827   ibmiotf.application.Client   INFO    Connected successfully: a:abc123:MyFirs
```

4.4.3 Connect a Device

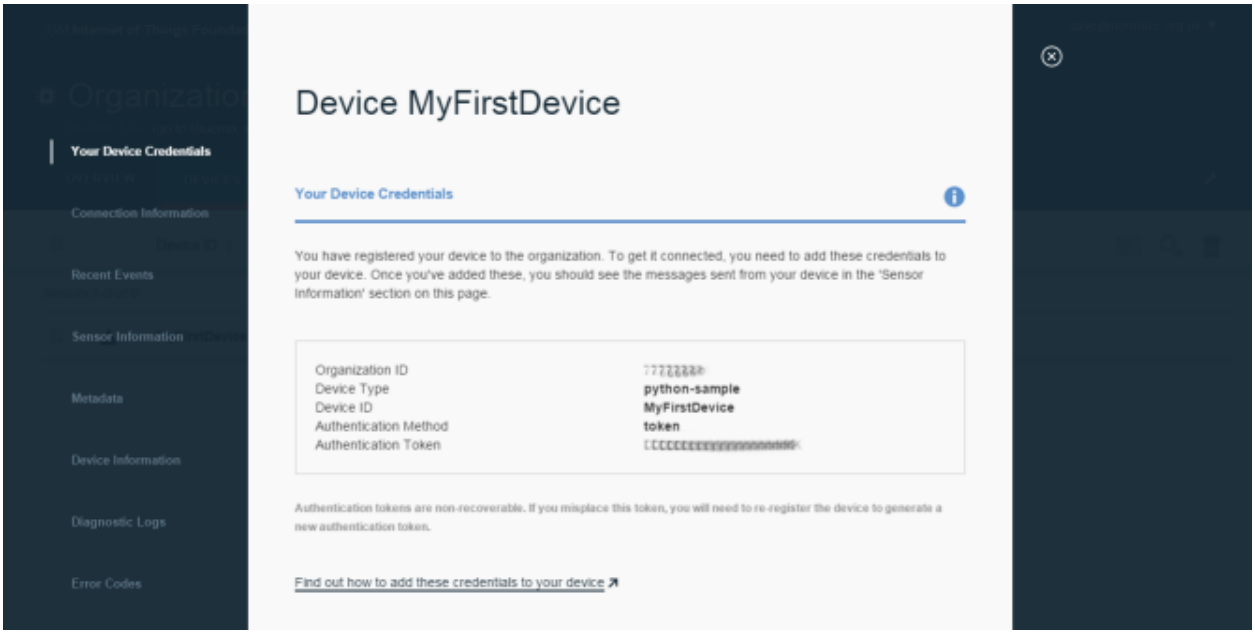
Before we can register a device we have to define a device type. A `device type` is intended to represent a group of devices that are identical, it can be useful to think of the `device type` as analogous to a model in the traditional model:serial identification scheme. Leave you application running to return to your Watson IoT platform Dashboard and navigate to `Devices > Device Types`, click `Create Type` and enter `python-sample` as the name, all other fields are optional, so we're just going to skip past the rest of the options for the device type for now.



Now it's time to register a device of this type, navigate to `Devices > Browse` and click `Add Device`, select the `python-sample` device type that we just created and click `Next`. On the second panel enter a device ID of "MyFirstDevice" and click `Next` to continue through the guided creation process until you reach the `Add` button.



Once you click `Add` your device is registered and you will be presented with a generated `authentication token` for that device.



We're now going to take that authentication token and write a totally minimal device client in python.

```
import time
import sys
import ibmiotf.application
import ibmiotf.device

deviceOptions = {
  "org": "<INSERT_ORG_ID>",
  "type": "python-sample",
  "id": "MyFirstDevice",
  "auth-method": "token",
  "auth-token": "<INSERT_AUTH_TOKEN>"
}

try:
  deviceCli = ibmiotf.device.Client(deviceOptions)
except Exception as e:
  print("Caught exception connecting device: %s" % str(e))
  sys.exit()

deviceCli.connect()
for x in range (0,10):
  data = { 'hello' : 'world', 'x' : x}
  deviceCli.publishEvent("greeting", "json", data)
  time.sleep(1)

deviceCli.disconnect()
```

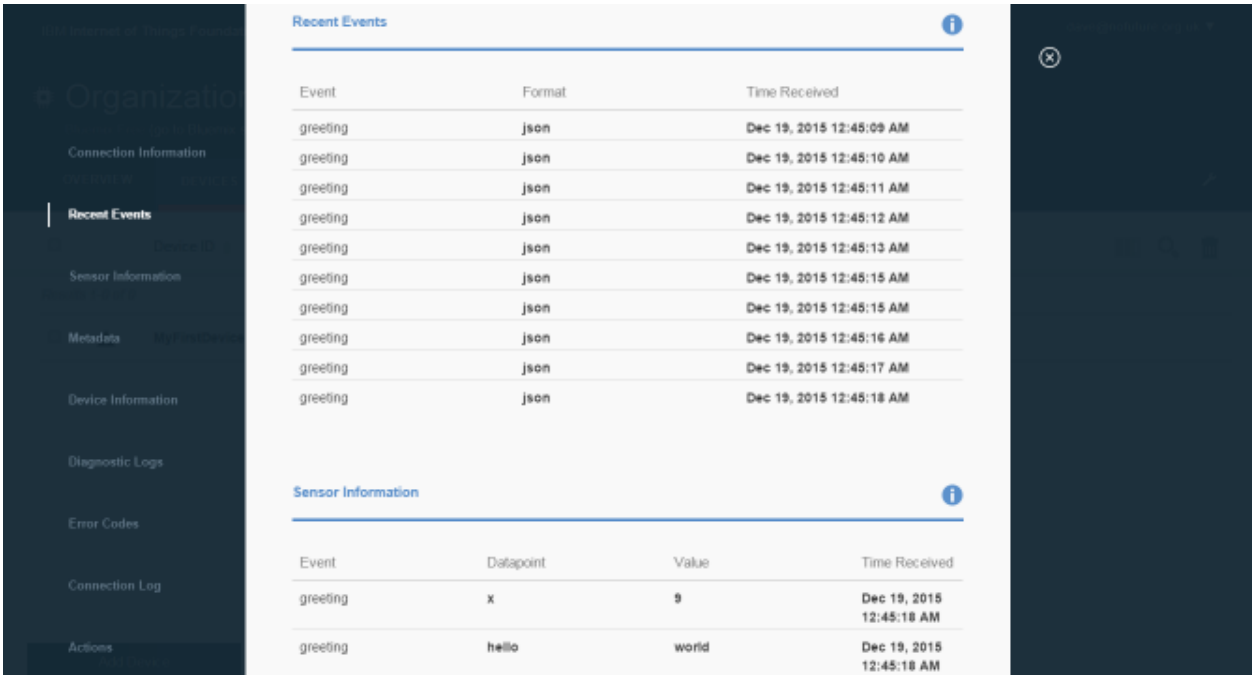
When executed the device code will generate 10 events .. code:

```
[user@host ~]$ python device.py
2015-12-19 00:34:57,675   ibmiotf.device.Client      INFO    Connected successfully: d:abc123:python-
2015-12-19 00:35:07,678   ibmiotf.device.Client      INFO    Disconnected from the IBM Internet of Th
2015-12-19 00:35:07,678   ibmiotf.device.Client      INFO    Messages published : 10, life: 11s, rate
2015-12-19 00:35:07,678   ibmiotf.device.Client      INFO    Messages received  : 0, life: 11s, rate:
2015-12-19 00:35:07,679   ibmiotf.device.Client      INFO    Closed connection to the IBM Internet of
```

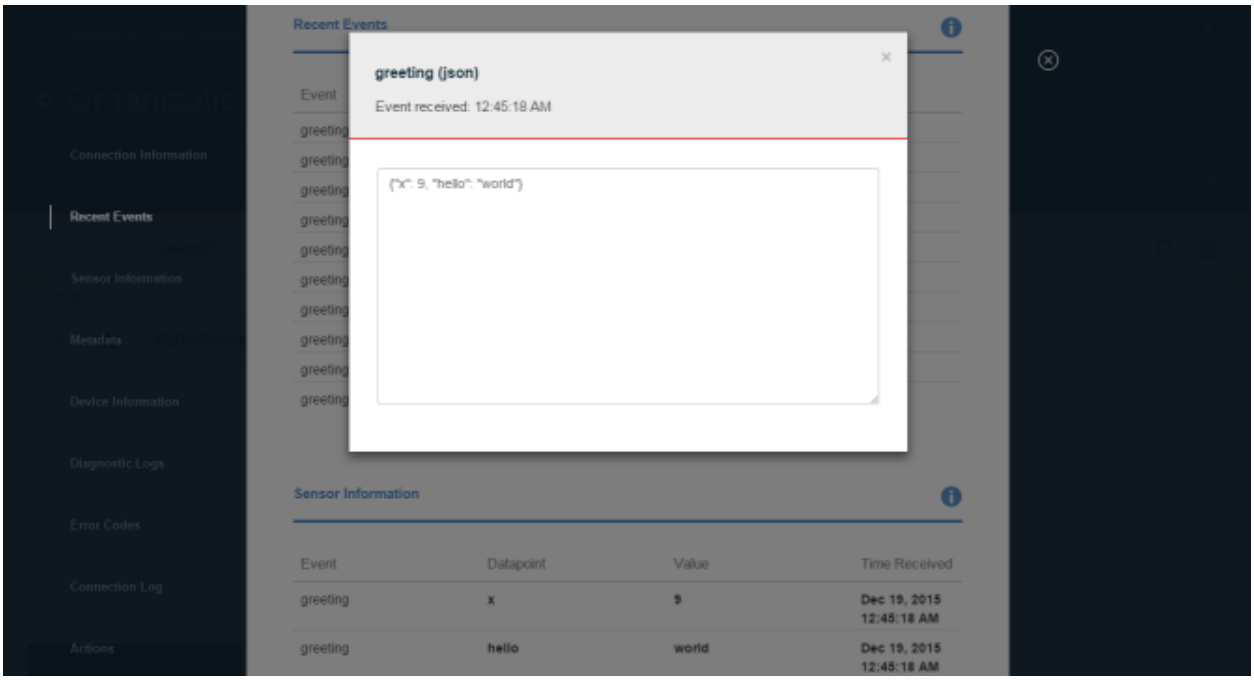
Returning to the terminal where your application is running you will be able to see the application is recieving and processing the events submitted by your device in real time.

```
[user@host ~]$ python test.py
(Press Ctrl+C to disconnect)
2015-12-19 00:34:27,865   ibmiotf.application.Client  INFO    Connected successfully: a:abc123:MyFirs
2015-12-19T00:34:57.687199+00:00 python-sample:MyFirstDevice     greeting: {"hello": "world", "x": 0}
2015-12-19T00:34:58.770336+00:00 python-sample:MyFirstDevice     greeting: {"hello": "world", "x": 1}
2015-12-19T00:34:59.686953+00:00 python-sample:MyFirstDevice     greeting: {"hello": "world", "x": 2}
2015-12-19T00:35:00.687080+00:00 python-sample:MyFirstDevice     greeting: {"hello": "world", "x": 3}
2015-12-19T00:35:01.687707+00:00 python-sample:MyFirstDevice     greeting: {"hello": "world", "x": 4}
2015-12-19T00:35:02.687834+00:00 python-sample:MyFirstDevice     greeting: {"hello": "world", "x": 5}
2015-12-19T00:35:04.393050+00:00 python-sample:MyFirstDevice     greeting: {"hello": "world", "x": 6}
2015-12-19T00:35:04.688588+00:00 python-sample:MyFirstDevice     greeting: {"hello": "world", "x": 7}
2015-12-19T00:35:05.689215+00:00 python-sample:MyFirstDevice     greeting: {"hello": "world", "x": 8}
2015-12-19T00:35:06.688842+00:00 python-sample:MyFirstDevice     greeting: {"hello": "world", "x": 9}
```

Finally, if you return to the Watson IoT platform dashboard and scroll down to `Device Events` you will be able to see all the events that you just sent.



You can also examine the content of any of the events by selecting an event from the table.


Recent Events
greeting (json)
Event received: 12:45:18 AM
{"x": 9, "hello": "world"}
Connection Information
Recent Events
Sensor Information
Metadata
Device Information
Diagnostic Logs
Error Codes
Connection Log
Actions
Sensor Information
Event
Datapoint
Value
Time Received
greeting
x
9
Dec 19, 2015 12:45:18 AM
greeting
hello
world
Dec 19, 2015 12:45:18 AM

CHAPTER 5

HTTP API for Applications

The IoT Platform API can be used to interact with your organization in the IoT Platform.

5.1 API Capabilities

The IoT Platform API supports the following functionality for applications:

- View organization details.
- Bulk device operations (list all, add, remove).
- Device type operations (list all, create, delete, view details, update).
- Device operations (list devices, add, remove, view details, update, view location, view management information).
- Device diagnostic operations (clear log, retrieve logs, add log information, delete logs, get specific log, clear error codes, get device error codes, add an error code).
- Connection problem determination (list device connection log events).
- Historical event retrieval (view events from all devices, view events from a device type, view events for a specific device).
- Device management request operations (list device management requests, initiate a request, clear request status, get details of a request, get list of request statuses for each affected device, get request status for a specific device).
- Usage management (retrieve number of active devices over a period of time, retrieve amount of storage used by historical event data, retrieve total amount of data used).
- Publish events on behalf of devices (beta)
- Service status queries (retrieve service statuses for an organization).

5.2 IoT Platform API Version 2

The current version of the IoT Platform API is version 2. We strongly recommend that all users build their solutions based on the version 2 APIs.

5.3 IoT Platform API Version 1

Version 1 of the API is still supported, however it should now be considered a deprecated API, many features listed above are not available in the version 1 API.

Note:

- Many of the API paths relating to devices have changed.
- Version 1 of the API does not support any device type operations, other than listing all device types.
- Operations related to device management, device management requests, diagnostic information, or location cannot be performed using version 1 of the API.

CHAPTER 6

MQTT Connectivity for Applications

6.1 Client connection

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

org_id.messaging.internetofthings.ibmcloud.com

6.1.1 Unencrypted client connection

Connect on port **1883**

Important: All information your application submits is being sent in plain text (including the API key and authentication token). We recommend the use of an encrypted connection whenever possible.

6.1.2 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 Platform 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)*.

6.2 MQTT client identifier

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

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.
- 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

---

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

6.3 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*

6.4 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 the IoT Platform. One way to get that data into the service would be to write an application that processes the data and publishes it to the IoT Platform.

6.5 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**

6.6 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

6.7 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

6.8 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

6.9 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.

6.10 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

6.11 Scalable Applications

You can build scalable applications which will load balance messages across multiple instances of your application by making a few changes to how your application connects to the IoT Platform. Applications taking advantage of this feature must only attempt to make non-durable subscriptions. A bit of experimentation may be needed to understand how many clients are needed for the optimum balance in load.

- Supply a client id of the form **A**:**org_id**:**app_id**
- **A** indicates the client is a scalable 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.
- Create a non-durable subscription

Note: Only non-durable subscriptions are supported for scalable applications. Please note that the client id must begin with a capital 'A' in order to designated as a scalable application by the IoT Platform. Multiple clients that are part of the scalable application should use the exact same client id.

6.11.1 How It Works

The IoT Platform service extends the MQTT 3.1.1 specification to provide support for shared subscriptions. Shared subscription can provide simple load balancing functionality for applications. A shared subscription might be needed if a back-end enterprise application can not keep up with the number of messages being published to a specific topic space. For example if many devices were publishing messages that are being processed by a single application. It might be helpful to leverage the load balancing capability of a shared subscription. IoT Platform shared subscription support is limited to non-durable subscriptions only.

A simple example of an auto-scaling application:

- client 1 connects as A:abc123:myApplication and subscribes to all device events client 1 will receive 100% of the device events published
- client 2 connects as A:abc123:myApplication and subscribes to all device events now, client 1 and client 2 will share all of the events published between them. that is the load is now shared between client 1 and client 2.
- client 3 connects as A:abc123:myApplication and subscribes to all device events now, instance 1, 2 and 3 will process the events shared amongst all three instances
- clients 2 and 3 unsubscribe from all device events now, although instance 2 and 3 are still connected to the service, instance 1 will be receiving all device events published

CHAPTER 7

Python for Application Developers

- See iot-python in GitHub
- See ibmiotf on PyPi

7.1 Constructor

The constructor builds the client instance, and accepts an options dict containing the following definitions:

- `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`).

If no options dict is provided, the client will connect to the IoT Platform Quickstart, and default to an unregistered device. The options dict creates definitions which are used to interact with the IoT Platform module.

```
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:
  ...
```

7.1.1 Using a configuration file

If you are not using an options dict as shown above, you include a configuration file containing an options dict. If you are using a configuration file containing an options dict, use the following code format.

```
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
```

7.2 API calls

Each method in the APIClient responds with either a valid response (JSON or boolean) in the case of success or IoTFCReSTException in the case of failure. The IoTFCReSTException contains the following properties that application can parse to get more information about the failure.

- httpcode - HTTP Status Code
- message - Exception message containing the reason for the failure
- response - JsonElement containing the partial response if any otherwise null

So in the case of failure, application needs to parse the response to see if the action is partially successful or not.

7.3 Subscribing to device events

Events are the mechanism by which devices publish data to the IoT Platform. 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 Platform 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.

By default, applications will subscribe to all events from all connected devices. Use the type, id, event and msgFormat parameters to control the scope of the subscription. A single client can support multiple subscriptions. The code samples below give examples of how to subscribe to devices dependent on device type, id, event and msgFormat parameters.

7.3.1 To subscribe to all events from all devices

```
import ibmiotf.application

options = ibmiotf.application.ParseConfigFile(configFilePath)
client = ibmiotf.application.Client(options)

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

7.3.2 To subscribe to all events from all devices of a specific type

```
import ibmiotf.application

options = ibmiotf.application.ParseConfigFile(configFilePath)
client = ibmiotf.application.Client(options)

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

7.3.3 To subscribe to a specific event from all devices

```
import ibmiotf.application

options = ibmiotf.application.ParseConfigFile(configFilePath)
client = ibmiotf.application.Client(options)

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

7.3.4 To subscribe to a specific event from two or more different devices

```
import ibmiotf.application

options = ibmiotf.application.ParseConfigFile(configFilePath)
client = ibmiotf.application.Client(options)

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

7.3.5 To subscribe to all events published by a device in json format

```
import ibmiotf.application

options = ibmiotf.application.ParseConfigFile(configFilePath)
client = ibmiotf.application.Client(options)

client.connect()
client.subscribeToDeviceEvents(deviceType=myDeviceType, deviceId=myDeviceId, msgFormat="json")
```

7.4 Handling events from devices

To process the events received by your subscriptions 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

```
import ibmiotf.application

options = ibmiotf.application.ParseConfigFile(configFilePath)
client = ibmiotf.application.Client(options)

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()
```

7.5 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.

7.5.1 Subscribe to status updates for all devices

```
import ibmiotf.application

options = ibmiotf.application.ParseConfigFile(configFilePath)
client = ibmiotf.application.Client(options)

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

7.5.2 Subscribe to status updates for all devices of a specific type

```
import ibmiotf.application

options = ibmiotf.application.ParseConfigFile(configFilePath)
client = ibmiotf.application.Client(options)

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

7.5.3 Subscribe to status updates for two different devices

```
import ibmiotf.application

options = ibmiotf.application.ParseConfigFile(configFilePath)
client = ibmiotf.application.Client(options)

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

7.6 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 - integer

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

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

```
import ibmiotf.application

options = ibmiotf.application.ParseConfigFile(configFilePath)
client = ibmiotf.application.Client(options)

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()
```

7.7 Publishing events from devices

Applications can publish events as if they originated from a Device

```
import ibmiotf.application

options = ibmiotf.application.ParseConfigFile(configFilePath)
client = ibmiotf.application.Client(options)

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

7.8 Publishing commands to devices

Applications can publish commands to connected devices

```
import ibmiotf.application

options = ibmiotf.application.ParseConfigFile(configFilePath)
client = ibmiotf.application.Client(options)

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

7.9 Organization details

Applications can use the `getOrganizationDetails()` method to retrieve the details about the configuration of the organization.

```
import ibmiotf.application

options = ibmiotf.application.ParseConfigFile(configFilePath)
client = ibmiotf.application.Client(options)

try:
    orgDetail = client.api.getOrganizationDetails()
except IoTFCReSTException as e:
    print("ERROR [" + e.httpcode + "] " + e.message)
```

Refer to the Organization Configuration section of the IBM IoT Platform API for information about the request & response model and http status code.

7.10 Bulk device operations

Applications can use bulk operations to get, add or remove devices in bulk.

Refer to the Bulk Operations section of the IBM IoT Platform API for information about the list of query parameters, the request & response model and http status code.

7.10.1 Retrieve device information

Bulk device information can be retrieved using the `getAllDevices()` method. This method retrieves information on all registered devices in the organization, each request can contain a maximum of 512KB.

The response contains parameters required by the application to retrieve the dictionary *results* from the response to get the array of devices returned. Other parameters in the response are required to make further calls, for example, the `_bookmark` element can be used to page through results. Issue the first request without specifying a bookmark, then take the bookmark returned in the response and provide it on the request for the next page. Repeat until the end of the result set indicated by the absence of a bookmark. Each request must use exactly the same values for the other parameters, or the results are undefined.

```
import ibmiotf.application

options = ibmiotf.application.ParseConfigFile(configFilePath)
client = ibmiotf.application.Client(options)

try:
    deviceList = client.api.getAllDevices()
except IoTFCReSTException as e:
    print("ERROR [" + e.httpcode + "] " + e.message)
```

7.10.2 Add Devices in bulk

The `addMultipleDevices()` method can be used to add one or more devices to your IoT Platform organization, the maximum size of a request is set to 512KB. The response will contain the generated authentication tokens for each added device. These authentication tokens must be recorded when processing the response, as lost authentication tokens cannot be retrieved.

```
import ibmiotf.application

options = ibmiotf.application.ParseConfigFile(configFilePath)
client = ibmiotf.application.Client(options)

listOfDevicesToAdd = [
    {'typeId' : "pi-model-a", 'deviceId' : '200020002004'},
    {'typeId' : "pi-model-b", 'deviceId' : '200020002005'}
]

try:
    deviceList = client.api.addMultipleDevices(listOfDevicesToAdd)
except IoTFCReSTException as e:
    print("ERROR [" + e.httpcode + "] " + e.message)
```

7.10.3 Delete Devices in bulk

The `deleteMultipleDevices()` method can be used to delete multiple devices from an IoT Platform organization, each request can contain a maximum of 512KB.

```
import ibmiotf.application

options = ibmiotf.application.ParseConfigFile(configFilePath)
client = ibmiotf.application.Client(options)

listOfDevicesToDelete = [
    {'typeId' : "pi-model-a", 'deviceId' : '200020002004'},
    {'typeId' : "pi-model-b", 'deviceId' : '200020002005'}
]

try:
    deviceList = client.api.deleteMultipleDevices(listOfDevicesToDelete)
except IoTFCReSTException as e:
    print("ERROR [" + e.httpcode + "] " + e.message)
```

7.11 Device Type Operations

Device types can be used as templates for adding device information to devices as they are added to your organization. Applications can use the IoT Platform API to list, create, delete, view, or update device types in your organization.

Refer to the Device Types section of the IBM IoT Platform API documentation for information about the list of query parameters, the request & response model, and http status codes.

7.11.1 Get all Device Types

The `getAllDeviceTypes()` method can be used to retrieve all device types in your IoT Platform organization. The response contains parameters and application needs to retrieve the dictionary *results* from the response to get the array of devices returned. Other parameters in the response are required to make further call, for example, the *_bookmark* element can be used to page through results. Issue the first request without specifying a bookmark, then take the bookmark returned in the response and provide it on the request for the next page. Repeat until the end of the result set indicated by the absence of a bookmark. Each request must use exactly the same values for the other parameters, or the results are undefined.

```
import ibmiotf.application

options = ibmiotf.application.ParseConfigFile(configFilePath)
client = ibmiotf.application.Client(options)

listOfDevicesToAdd = [
    {'typeId' : "pi-model-a", 'deviceId' : '200020002004'},
    {'typeId' : "pi-model-b", 'deviceId' : '200020002005'}
]

try:
    options = {'_limit' : 2}
    deviceTypeList = client.api.getAllDeviceTypes(options)
except IoTFCReSTException as e:
    print("ERROR [" + e.httpcode + "] " + e.message)
```

7.11.2 Add a Device Type

The `addDeviceType()` method can be used to register a device type to IoT Platform. In each request, you must first define the device information, and device metadata elements which you want to be applied to all devices of this type. The device information element is comprised of several variables, including, serial number, manufacturer, model, class, description, firmware and hardware versions, and descriptive location. The metadata element is comprised of custom variables and values which can be defined by the user.

```
import ibmiotf.application

options = ibmiotf.application.ParseConfigFile(configFilePath)
client = ibmiotf.application.Client(options)

info = {
    "serialNumber": "100087",
    "manufacturer": "ACME Co.",
    "model": "7865",
    "deviceClass": "A",
    "description": "My shiny device",
    "fwVersion": "1.0.0",
    "hwVersion": "1.0",
    "descriptiveLocation": "Office 5, D Block"
}
meta = {
    "customField1": "customValue1",
    "customField2": "customValue2"
}

try:
    deviceType = client.api.addDeviceType(deviceType = "myDeviceType", description = "My first device
except IoTFCReSTException as e:
    print("ERROR [" + e.httpcode + "] " + e.message)
```

7.11.3 Delete a Device Type

The `deleteDeviceType()` method can be used to delete a device type from your IoT Platform organization.

```
import ibmiotf.application

options = ibmiotf.application.ParseConfigFile(configFilePath)
client = ibmiotf.application.Client(options)

try:
    success = client.api.deleteDeviceType("myDeviceType")
except IoTFCReSTException as e:
    print("ERROR [" + e.httpcode + "] " + e.message)
```

7.11.4 Get a Device Type

The `getDeviceType()` method retrieves information on a given device type. The `typeId` of the device type you wish to retrieve must be used as a parameter

```
import ibmiotf.application

options = ibmiotf.application.ParseConfigFile(configFilePath)
```

```
client = ibmiotf.application.Client(options)

try:
    deviceTypeInfo = client.api.getDeviceType("myDeviceType")
except IoTFCReSTException as e:
    print("ERROR [" + e.httpcode + "] " + e.message)
```

7.11.5 Update a Device Type

The `updateDeviceType()` method can be used to modify the properties of a device type. When using this method, several parameters must be defined. Firstly, the `typeId` of the device type to be updated must be specified, then the `description`, `deviceInfo`, and `metadata` elements.

```
import ibmiotf.application

options = ibmiotf.application.ParseConfigFile(configFilePath)
client = ibmiotf.application.Client(options)

info = {
    "serialNumber": "100087",
    "manufacturer": "ACME Co.",
    "model": "7865",
    "deviceClass": "A",
    "description": "My shiny device",
    "fwVersion": "1.0.0",
    "hwVersion": "1.0",
    "descriptiveLocation": "Office 5, D Block"
}
meta = {
    "customField1": "customValue1",
    "customField2": "customValue2",
    "customField3": "customValue3"
}

try:
    updatedDeviceTypeInfo = client.api.updateDeviceType("myDeviceType", "New description", deviceInfo
except IoTFCReSTException as e:
    print("ERROR [" + e.httpcode + "] " + e.message)
```

7.12 Device operations

Device operations made available through the API include listing, adding, removing, viewing, updating, viewing location and viewing device management information of devices in an IoT Platform organization.

Refer to the Device section of the IoT Platform API for information about the list of query parameters, the request & response model and http status code.

7.12.1 Get Devices of a particular Device Type

The *retrieveDevices()* method can be used to retrieve all the devices of a particular device type in an organization from IoT Platform. For example,

```
print("\nRetrieving All existing devices")
print("Retrieved Devices = ", apiCli.retrieveDevices(deviceTypeId))
```

The response contains parameters and application needs to retrieve the dictionary *results* from the response to get the array of devices returned. Other parameters in the response are required to make further call, for example, the *_bookmark* element can be used to page through results. Issue the first request without specifying a bookmark, then take the bookmark returned in the response and provide it on the request for the next page. Repeat until the end of the result set indicated by the absence of a bookmark. Each request must use exactly the same values for the other parameters, or the results are undefined.

In order to pass the *_bookmark* or any other condition, the overloaded method must be used. The overloaded method takes the parameters in the form of dictionary as shown below,

```
response = apiClient.retrieveDevices("iotsample-ardunio", parameters);
```

The above snippet sorts the response based on device id and uses the bookmark to page through the results.

7.12.2 Add a Device

The *registerDevice()* method is used to add a device to an IoT Platform organization. The *registerDevice()* method adds a single device to your IoT Platform organization. The parameters which can be set when adding a device are:

- deviceTypeId: *Optional*. Assigns a device type to the device. Where there is a clash between variables defined by the device type and variables defined by under deviceInfo, the device specific variables will take precedence.
- deviceId: *Mandatory*.
- authToken: *Optional*. If no authentication token is supplied, one will be generated and included in the response.
- deviceInfo: *Optional*. This parameter is optional, and can contain a number of variables, including: serialNumber, manufacturer, model, deviceClass, description, firmware and hardware versions, and descriptiveLocation.
- metadata: *Optional*. Metadata can optionally be added in the form of custom field:value string pairs. An example is given in the code sample below.
- location: *Optional*. This parameter contains the longitude, latitude, elevation, accuracy, and mesauredDateTime variables.

For more information on the parameters presented here, and the response format and codes, please see the relevant section of API documentation.

When using the *registerDevice()* method, you must define the mandatory deviceID parameter, and any of the optional parameters you require for your device, then call the method using the parameters you've selected.

7.12.3 Sample

The following code sample should be inserted after the constructor code in a .py file. This code demonstrates defining the deviceId, authToken, metadata, deviceInfo parameters, and location parameters and then using the method with those parameters and adding a device type.

```
deviceId = "200020002000"
authToken = "password"
metadata = {"customField1": "customValue1", "customField2": "customValue2"}
deviceInfo = {"serialNumber": "001", "manufacturer": "Blueberry", "model": "abc1", "deviceClass": "A
location = {"longitude" : "12.78", "latitude" : "45.90", "elevation" : "2000", "accuracy" : "0", "mea

apiCli.registerDevice(deviceTypeId, deviceId, metadata, deviceInfo, location)
```

7.12.4 Delete a Device

The *deleteDevice()* method is used to remove a device from an IoT Platform organization. When deleting a device using this method, the parameters which must be specified in the method are the deviceTypeId, and the deviceId.

The following code snippet provides an example of the format required for this method.

```
apiCli.deleteDevice(deviceTypeId, deviceId)
```

7.12.5 Get a Device

The *getDevice()* method can be used to retrieve a device from an IoT Platform organization. When retrieving device details using this method, the parameters which must be specified in the method are the deviceTypeId, and the deviceId.

The following code snippet provides an example of the format required for this method.

```
apiCli.getDevice(deviceTypeId, deviceId)
```

7.12.6 Get all Devices

The *getAllDevices()* method can be used to retrieve all devices within an IoT Platform organization.

```
apiCli.getAllDevices({'typeId' : deviceTypeId})
```

7.12.7 Update a Device

The *updateDevice()* method is used to modify one or more properties of a device. Any property in the deviceInfo or metadata parameters can be updated. In order to update a device property, it must be defined above the method. The status parameter should contain "alert": True. The Alert property controls whether a device will display error codes in the IoT Platform user interface, and should be set by default to 'True'.

```
status = { "alert": { "enabled": True }  }
apiCli.updateDevice(deviceTypeId, deviceId, metadata2, deviceInfo, status)
```

7.12.8 Sample

In this sample, the following code identifies a specific device, and updates several properties under the deviceInfo parameter.

```
    status = { "alert": { "enabled": True } }
    deviceInfo = {descriptiveLocation: "London", hwVersion: "2.0.1", fwVersion: "2.5.1"}
apiCli.updateDevice("MyDeviceType", "200020002000", deviceInfo, status)
```

7.12.9 Get Location Information

The *getDeviceLocation()* method can be used to retrieve the location information of a device. The parameters required for retrieving the location data are deviceTypeId and deviceId.

```
apiClient.getDeviceLocation("iotsample-ardunio", "ardunio01")
```

The response to this method contains the longitude, latitude, elevation, accuracy, measuredTimeStamp, and updated-TimeStamp properties.

7.12.10 Update Location Information

The *updateDeviceLocation()* method is used to modify the location information for a device. Simlarly to updating device properties, the deviceLocation parameter must be defined with the changes you wish to apply. The code sample below demonstrates changing the location data for a given device.

```
deviceLocation = { "longitude": 0, "latitude": 0, "elevation": 0, "accuracy": 0, "measuredDateTime":
apiCli.updateDeviceLocation(deviceTypeId, deviceId, deviceLocation)
```

If no date is supplied, the entry is added with the current date and time.

7.12.11 Get Device Management Information

The *getDeviceManagementInformation()* method is used to get the device management information for a device. The response contains the last activity date-time, the device's dormant status (true/false), support for device and firmware actions, and firmware data. For a comprehensive list of response content, please see the relevant API documentation.

The following code sample will return the device management information for a device with the deviceId "00aabbccde03", with deviceTypeId "iotsample-arduino".

7.12.12 Sample

```
apiCli.getDeviceManagementInformation("iotsample-arduino", "00aabbccde03")
```

7.13 Device diagnostic operations

Applications can use device diagnostic operations to clear logs, retrieve all or specific logs for a device, add log information, delete logs, clear error codes, get device error codes, and add an error codes.

For more detailed information on query and response models, response codes, and query paramters, please see the relevant API documentation.

7.13.1 Get Diagnostic logs

The *getAllDiagnosticLogs()* method is used to retrieve all diagnostic logs for a specific device. The *getAllDiagnosticLogs()* method requires the deviceTypeId and deviceId parameters.

```
apiCli.getAllDiagnosticLogs(deviceTypeId, deviceId)
```

The response model for this method contains the logId, message, severity, data, and timestamp.

7.13.2 Clear Diagnostic logs for a Device

The *clearAllDiagnosticLogs()* method is used to delete all diagnostic logs for a specific device. The required parameters are deviceTypeId and deviceId. Care should be taken when deleting logs, as logs cannot be recovered once deleted.

```
apiCli.clearAllDiagnosticLogs(deviceTypeId, deviceId)
```

7.13.3 Add a Diagnostic log

The *addDiagnosticLog()* method is used to add an entry in the diagnostic log of the device. The log may be pruned as the new entry is added. If no date is supplied, the entry is added with the current date and time. To use this method, first define a 'logs' parameter with the following variables:

- message: This variable is mandatory, and contains the new diagnostic message.
- severity: This variable is optional. If used it corresponds to the severity of the diagnostic log, and should be 0, 1, or 2, corresponding to the informational, warning, and error categories.
- data: This variable is optional, and should contain diagnostic data.
- timestamp: This variable is optional, and contains the date and time of the log entry in ISO8601 format. If this variable is not included, it is automatically added with the current date and time.

The other necessary paramteres required in the method are the deviceTypeId and deviceId for the specific device.

The code sample below contains an example of the method.

```
logs = { "message": "MessageContent", "severity": 0, "data": "LogData"}
apiCli.addDiagnosticLog(deviceTypeId, deviceId, logs)
```

7.13.4 Retrieve a specific Diagnostic log

The *getDiagnosticLog()* method is used to retrieve a specific diagnostic log for a specified device based on the log id. The required parameters for this method are the deviceTypeId, deviceId, and logId.

```
apiCli.getDiagnosticLog(deviceTypeId, deviceId, logId)
```

7.13.5 Delete a Diagnostic log

The *deleteDiagnosticLog()* can be used to delete a specific diagnostic log. In order to specify a diagnostic log, the deviceTypeId, deviceId, and logId parameters should be supplied.

```
apiCli.deleteDiagnosticLog(deviceTypeId, deviceId, logId)
```

7.13.6 Retrieve Device Error Codes

The *getAllDiagnosticErrorCodes()* method is used to retrieve all diagnostic error codes associated with a specific device.

```
apiCli.getAllDiagnosticErrorCodes(deviceTypeId, deviceId)
```

7.13.7 Clear Diagnostic Error Codes

The *clearAllErrorCodes()* method is used to clear the list of error codes associated with the device. The list is replaced with a single error code of zero.

```
apiCli.clearAllErrorCodes(deviceTypeId, deviceId)
```

7.13.8 Add single Diagnostic ErrorCode

The *addErrorCode()* method is used to add an error code to the list of error codes associated with the device. The list may be pruned as the new entry is added. The parameters required in the method are deviceTypeId, deviceId, and errorCode. The errorCode parameter contains the following variables:

- errorCode: This variable is mandatory and should be set as an integer. This sets the number of the error code to be created.
- timestamp: This variable is optional, and contains the date and time of the log entry in ISO8601 format. If this variable is not included, it is automatically added with the current date and time.

```
errorCode = { "errorCode": 1234, "timestamp": "2015-10-29T05:43:57.112Z" }
apiCli.addErrorCode(deviceTypeId, deviceId, errorCode)
```

7.14 Connection problem determination

The *getDeviceConnectionLogs()* method is used to list connection log events for a device. This information can be used to help diagnose connectivity problems between the device and the IoT Platform service. The entries record successful connection, unsuccessful connection attempts, intentional disconnection and server-initiated disconnection events.

```
apiCli.getDeviceConnectionLogs(deviceTypeId, deviceId)
```

The response includes a list of log entries, containing log messages and timestamps.

7.15 Historical Event Retrieval

These operations can be used to view events from all devices, view events from a device type or to view events for a specific device.

Refer to the Historical Event Retrieval section of the IBM IoT Platform API for information about the list of query parameters, the request & response model and http status code.

7.15.1 View events from all devices

Method getHistoricalEvents() can be used to view events across all devices registered to the organization.

```
print("Historical Events = ", apiCli.getHistoricalEvents())
```

The response will contain some parameters and the application needs to retrieve the JSON element *events* from the response to get the array of events returned. Other parameters in the response are required to make further call, for example, the *_bookmark* element can be used to page through results. Issue the first request without specifying a bookmark, then take the bookmark returned in the response and provide it on the request for the next page. Repeat until the end of the result set indicated by the absence of a bookmark. Each request must use exactly the same values for the other parameters, or the results are undefined.

In order to pass the *_bookmark* or any other condition, the overloaded method must be used. The overloaded method takes the parameters in the form of dictionary as shown below,

```
 startTime = math.floor(time.mktime((2013, 10, 10, 17, 3, 38, 0, 0, 0)) * 1000)
 endTime =  math.floor(time.mktime((2015, 10, 29, 17, 3, 38, 0, 0, 0)) * 1000)
 duration = {'start' : startTime, 'end' : endTime }
apiCli.getHistoricalEvents(options = duration))
```

The above snippet returns the events between the start and end time.

7.15.2 View events from a device type

The *getHistoricalEvents()* method is used to view events from all the devices of a particular device type.

```
apiCli.getHistoricalEvents(deviceType = 'iotsample-arduino', options = duration)
```

The response will contain some parameters and the application needs to retrieve the JSON element *events* from the response to get the array of events returned. As mentioned in the *view events from all devices* section, the overloaded method can be used to control the output.

7.15.3 View events from a device

The *getHistoricalEvents()* method is used to view events from a specific device. DeviceTypeId and deviceId parameters are required in order to use this method.

```
apiCli.getHistoricalEvents(deviceType, deviceId, options = duration)
```

The response will contain more parameters and application needs to retrieve the JSON element *events* from the response to get the array of events returned.

CHAPTER 8

Node.js for Application Developers

- See iot-nodejs in GitHub
- See the samples for device in Github
- See ibmiotf on NPM

CHAPTER 9

Application

ApplicationClient is application client for the Internet of Things Foundation service. This section contains information on how applications interact with devices.

9.1 Constructor

The constructor builds the application client instance. It accepts an configuration json containing the following :

- org - Your organization ID
- id - The unique ID of your application within your organization.
- auth-key - API key
- auth-token - API key token
- type - use 'shared' to enable shared subscription

If you want to use quickstart, then send only the first two properties.

```
var Client = require("ibmiotf");
    var appClientConfig = {
            "org" : orgId,
            "id" : appId,
            "auth-key" : apiKey,
            "auth-token" : apiToken
    }

    var appClient = new Client.IotfApplication(appClientConfig);

    .....
```

9.1.1 Using a configuration file

Instead of passing the configuration json directly, you can also use a configuration file. Use the following code snippet

```
var Client = require("ibmiotf");
    var appClientConfig = require("./application.json");

    var appClient = new Client.IotfApplication(appClientConfig);

    .....
```

The configuration file must be in the format of

```
{
      "org": 'xxxxx',
      "id": 'myapp',
      "auth-key": 'a-xxxxxxx-zenkqyfiea',
      "auth-token": 'xxxxxxxxxx'
}
```

9.2 Connecting to the IoT Platform

Connect to the IBM Watson Internet of Things Platform by calling the *connect*

```
var appClient = new Client.IotfApplication(appClientConfig);

    appClient.connect();

    appClient.on("connect", function () {

    //Add your code here
    });
```

After the successful connection to the IoT Platform service, the application client sends a *connect* event. So all the logic can be implemented inside this callback function.

9.3 Logging

By default, all the logs of `'warn'` are logged. If you want to enable more logs, use the *log.setLevel* function. Supported log levels - *trace, debug, info, warn, error*.

```
var appClient = new Client.IotfApplication(appClientConfig);

appClient.connect();
//setting the log level to 'trace'
appClient.log.setLevel('trace');
appClient.on("connect", function () {

//Add your code here
});
```

9.4 Shared Subscription

Use this feature to build scalable applications which will load balance messages across multiple instances of the application. To enable this, pass 'type' as 'shared' in the configuration.

```
var appClientConfig = {
  org: 'xxxxx',
  id: 'myapp',
  "auth-key": 'a-xxxxxx-xxxxxxxxx',
  "auth-token": 'xxxxx!xxxxxxxx',
  "type" : "shared" // make this connection as shared subscription
```

```
};
var appClient = new Client.IotfApplication(appClientConfig);

appClient.connect();
//setting the log level to 'trace'
appClient.log.setLevel('trace');
appClient.on("connect", function () {

//Add your code here
});
```

9.5 Handling errors

When the application clients encounters an error, it emits an *error* event.

```
var appClient = new Client.IotfApplication(appClientConfig);

appClient.connect();
//setting the log level to 'trace'
appClient.log.setLevel('trace');
appClient.on("connect", function () {

//Add your code here
});
appClient.on("error", function (err) {
        console.log("Error : "+err);
});
```

9.6 Subscribing to device events

Events are the mechanism by which devices publish data to the IoT Platform. 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 Platform, the credentials of the connection on which the event was received are used to determine from which device the event was sent. Using this architecture, it is impossible for a device to impersonate another device.

By default, applications will subscribe to all events from all connected devices. Use the type, id, event and msgFormat parameters to control the scope of the subscription. A single client can support multiple subscriptions. The code samples below give examples of how to subscribe to devices dependent on device type, id, event and msgFormat parameters.

9.6.1 To subscribe to all events from all devices

```
var appClient = new Client.IotfApplication(appClientConfig);

appClient.connect();

appClient.on("connect", function () {

        appClient.subscribeToDeviceEvents();
});
```

9.6.2 To subscribe to all events from all devices of a specific type

```
var appClient = new Client.IotfApplication(appClientConfig);

appClient.connect();

appClient.on("connect", function () {

        appClient.subscribeToDeviceEvents("mydeviceType");
});
```

9.6.3 To subscribe to a specific event from all devices

```
var appClient = new Client.IotfApplication(appClientConfig);

appClient.connect();

appClient.on("connect", function () {

        appClient.subscribeToDeviceEvents("+","+","myevent");
});
```

9.6.4 To subscribe to a specific event from two or more different devices

```
var appClient = new Client.IotfApplication(appClientConfig);

appClient.connect();

appClient.on("connect", function () {

        appClient.subscribeToDeviceEvents("myDeviceType","device01","myevent");
        appClient.subscribeToDeviceEvents("myOtherDeviceType","device02","myevent");
});
```

9.6.5 To subscribe to all events published by a device in json format

```
var appClient = new Client.IotfApplication(appClientConfig);

appClient.connect();

appClient.on("connect", function () {

        appClient.subscribeToDeviceEvents("myDeviceType","device01","+","json");

});
```

9.6.6 Handling events from devices

To process the events received by your subscriptions you need to implement a device event callback method. The ibmiotf application client emits the event *deviceEvent*. This function has the following properties:

- deviceType

- deviceId
- eventType
- format
- payload - Device event payload
- topic - Original topic

```
var appClient = new Client.IotfApplication(appClientConfig);

appClient.connect();

appClient.on("connect", function () {

        appClient.subscribeToDeviceEvents("myDeviceType","device01","+","json");

});
appClient.on("deviceEvent", function (deviceType, deviceId, eventType, format, payload) {

        console.log("Device Event from :: "+deviceType+" : "+deviceId+" of event "+eventType+" with p

});
```

9.7 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.

9.7.1 Subscribe to status updates for all devices

```
var appClient = new Client.IotfApplication(appClientConfig);

appClient.connect();

appClient.on("connect", function () {

        appClient.subscribeToDeviceStatus();

});
```

9.7.2 Subscribe to status updates for all devices of a specific type

```
var appClient = new Client.IotfApplication(appClientConfig);

appClient.connect();

appClient.on("connect", function () {

        appClient.subscribeToDeviceStatus("myDeviceType");

});
```

9.7.3 Subscribe to status updates for two different devices

```
var appClient = new Client.IotfApplication(appClientConfig);

appClient.connect();

appClient.on("connect", function () {

        appClient.subscribeToDeviceStatus("myDeviceType","device01");
        appClient.subscribeToDeviceStatus("myOtherDeviceType","device02");

});
```

9.7.4 Handling status updates from devices

To process the status updates received by your subscriptions you need to implement an device status callback method. The ibmiotf application client emits the event *deviceStatus*. This function has the following properties:

- deviceType
- deviceId
- payload - Device status payload
- topic

```
var appClient = new Client.IotfApplication(appClientConfig);

appClient.connect();

appClient.on("connect", function () {

        appClient.subscribeToDeviceStatus("myDeviceType","device01");
        appClient.subscribeToDeviceStatus("myOtherDeviceType","device02");

});
appClient.on("deviceStatus", function (deviceType, deviceId, payload, topic) {

        console.log("Device status from :: "+deviceType+" : "+deviceId+" with payload : "+payload);

});
```

9.8 Publishing events from devices

Applications can publish events as if they originated from a Device. The function requires:

- DeviceType
- Device ID
- Event Type
- Format
- Data

```
var appClient = new Client.IotfApplication(appClientConfig);

appClient.connect();

appClient.on("connect", function () {

        var myData={'name' : 'foo', 'cpu' : 60, 'mem' : 50};
        myData = JSON.stringify(myData);
        appClient.publishDeviceEvent("myDeviceType","device01", "myEvent", "json", myData);

});
```

9.9 Publishing commands to devices

Applications can publish commands to connected devices. The function requires:

- DeviceType
- Device ID
- Command Type
- Format
- Data

```
var appClient = new Client.IotfApplication(appClientConfig);

appClient.connect();

appClient.on("connect", function () {

        var myData={'DelaySeconds' : 10};
        myData = JSON.stringify(myData);
        appClient.publishDeviceCommand("myDeviceType","device01", "reboot", "json", myData);

});
```

9.10 Disconnect Client

Disconnects the client and releases the connections

```
var appClient = new Client.IotfApplication(appClientConfig);

appClient.connect();

appClient.on("connect", function () {

        var myData={'DelaySeconds' : 10}
        appClient.publishDeviceCommand("myDeviceType","device01", "reboot", "json", myData);

        appClient.disconnect();
});
```

CHAPTER 10

Java for Application Developers

- See iot-java in GitHub

10.1 Constructor

The constructor builds the client instance, and accepts a Properties object containing the following definitions:

- `org` - Your organization ID (This is a required field. In case of quickstart flow, provide org as `quickstart`).
- `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`).
- `clean-session` - true or false (required only if you want to connect the application in durable subscription. By default the clean-session is set to true).
- `shared-subscription` - true or false (required only if shared subscription needs to be enabled).

Note: One must set `shared-subscription` to true to build scalable applications which will load balance messages across multiple instances of the application. Refer to the scalable applications section for more information about the load balancing.

The Properties object creates definitions which are used to interact with the IoT Platform module. If no options are provided or organization is provided as quickstart, the client will connect to the IoT Platform Quickstart, and default to an unregistered device.

The following code snippet shows how to construct the ApplicationClient instance in Quickstart mode,

```
import com.ibm.iotf.client.app.ApplicationClient;
import java.util.Properties;

Properties options = new Properties();
options.put("org", "quickstart");

ApplicationClient myClient = new ApplicationClient(options);
```

The following code snippet shows how to construct the ApplicationClient instance in registered flow,

```
Properties options = new Properties();
options.put("org", "uguhsp");
options.put("id", "app" + (Math.random() * 10000));
options.put("Authentication-Method","apikey");
options.put("API-Key", "<API-Key>");
options.put("Authentication-Token", "<Authentication-Token>");

ApplicationClient myClient = new ApplicationClient(options);
```

10.1.1 Using a configuration file

Instead of including a Properties object directly, you can use a configuration file containing the name-value pairs for Properties. If you are using a configuration file containing a Properties object, use the following code format.

```
Properties props = ApplicationClient.parsePropertiesFile(new File("C:\\temp\\application.prop"));
ApplicationClient myClient = new ApplicationClient(props);
...
```

The application configuration file must be in the following format:

```
[application]
org=$orgId
id=$myApplication
auth-method=apikey
auth-key=$key
auth-token=$token
enable-shared-subscription=true|false
```

10.2 Connecting to the IoT Platform

Connect to the IoT Platform by calling the *connect* function.

```
Properties props = ApplicationClient.parsePropertiesFile(new File("C:\\temp\\application.prop"));
ApplicationClient myClient = new ApplicationClient(props);

myClient.connect();
```

After the successful connection to the IoT Platform service, the application client can perform the following operations, like subscribing to device events, subscribing to device status, publishing device events and commands.

10.3 Subscribing to device events

Events are the mechanism by which devices publish data to the IoT Platform. 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 Platform 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.

By default, applications will subscribe to all events from all connected devices. Use the type, id, event and msgFormat parameters to control the scope of the subscription. A single client can support multiple subscriptions. The code samples below give examples of how to subscribe to devices dependent on device type, id, event and msgFormat parameters.

10.3.1 To subscribe to all events from all devices

```
myClient.connect();
myClient.subscribeToDeviceEvents();
```

10.3.2 To subscribe to all events from all devices of a specific type

```
myClient.connect();
myClient.subscribeToDeviceEvents("iotsample-ardunio");
```

10.3.3 To subscribe to all events from a specific device

```
myClient.connect();
myClient.subscribeToDeviceEvents("iotsample-ardunio", "00aabbccddee");
```

10.3.4 To subscribe to a specific event from two or more different devices

```
myClient.connect();
myClient.subscribeToDeviceEvents("iotsample-ardunio", "00aabbccddee", "myEvent");
myClient.subscribeToDeviceEvents("iotsample-ardunio", "10aabbccddee", "myEvent");
```

10.3.5 To subscribe to events published by a device in json format

```
client.connect()
myClient.subscribeToDeviceEvents("iotsample-ardunio", "00aabbccddee", "myEvent", "json", 0);
```

10.4 Handling events from devices

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

- 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

A sample implementation of the Event callback,

```
import com.ibm.iotf.client.app.Event;
import com.ibm.iotf.client.app.EventCallback;
import com.ibm.iotf.client.app.Command;

public class MyEventCallback implements EventCallback {
    public void processEvent(Event e) {
        System.out.println("Event:: " + e.getDeviceId() + ":" + e.getEvent() + ":" + e.getPayload());
    }

    public void processCommand(Command cmd) {
        System.out.println("Command " + cmd.getPayload());
    }
}
```

Once the event callback is added to the ApplicationClient, the processEvent() method is invoked whenever any event is published on the subscribed criteria, The following snippet shows how to add the Event call back into Application-Client instance,

```
myClient.connect()
myClient.setEventCallback(new MyEventCallback());
myClient.subscribeToDeviceEvents();
```

Similar to subscribing to device events, the application can subscribe to commands that are sent to the devices. Following code snippet shows how to subscribe to all commands to all the devices in the organization:

```
myClient.connect()
myClient.setEventCallback(new MyEventCallback());
myClient.subscribeToDeviceCommands();
```

Overloaded methods are available to control the command subscription. The processCommand() method is called when a command is sent to the device that matches the command subscription.

10.5 Subscribing to device status

Similar to subscribing to device events, applications can subscribe to device status, like device connect and disconnect to IoT Platform. By default, this will subscribe to status updates for all connected devices. Use the Device Type and Device Id parameters to control the scope of the subscription. A single ApplicationClient can support multiple subscriptions.

10.5.1 Subscribe to status updates for all devices

```
myClient.connect();
myClient.subscribeToDeviceStatus();
```

10.5.2 Subscribe to status updates for all devices of a specific type

```
myClient.connect();
myClient.subscribeToDeviceStatus("iotsample-ardunio");
```

10.5.3 Subscribe to status updates for two different devices

```
myClient.connect();
myClient.subscribeToDeviceStatus("iotsample-ardunio", "00aabbccddee");
myClient.subscribeToDeviceStatus("iotsample-ardunio", "10aabbccddee");
```

10.6 Handling status updates from devices

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

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 - java.util.Date
- status.action - string
- status.connectTime - java.util.Date
- status.port - integer

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

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

A sample implementation of the Status callback,

```
private static class MyStatusCallback implements StatusCallback {

    public void processApplicationStatus(ApplicationStatus status) {
        System.out.println("Application Status = " + status.getPayload());
    }

    public void processDeviceStatus(DeviceStatus status) {
        if(status.getAction() == "Disconnect") {
            System.out.println("device: "+status.getDeviceId()
                               + "  time: "+ status.getTime()
                               + "  action: " + status.getAction()
                               + "  reason: " + status.getReason());
        } else {
            System.out.println("device: "+status.getDeviceId()
                               + "  time: "+ status.getTime()
                               + "  action: " + status.getAction());
        }
```

```
    }
}
```

Once the status callback is added to the ApplicationClient, the processDeviceStatus() method is invoked whenever any device is connected or disconnected from IoT Platform that matches the criteria, The following snippet shows how to add the status call back instance into ApplicationClient,

```
myClient.connect()
myClient.setStatusCallback(new MyStatusCallback());
myClient.subscribeToDeviceStatus();
```

As similar to device status, the application can subscribe to any other application connect or disconnect status as well. Following code snippet shows how to subscribe to the application status in the organization:

```
myClient.connect()
myClient.setEventCallback(new MyEventCallback());
myClient.subscribeToApplicationStatus();
```

Overloaded method is available to control the status subscription to a particular application. The processApplication-Status() method is called whenever any application is connected or disconnected from IoT Platform that matches the criteria.

10.7 Publishing events from devices

Applications can publish events as if they originated from a Device.

```
myClient.connect()

//Generate the event to be published
JsonObject event = new JsonObject();
event.addProperty("name", "foo");
event.addProperty("cpu",  60);
event.addProperty("mem",  40);

// publish the event on behalf of device
myClient.publishEvent(deviceType, deviceId, "blink", event);
```

10.7.1 Publish events using HTTP(s)

Apart from MQTT, the application can publish device events to IBM Watson IoT Platform using HTTP(s) by following 3 simple steps,

- Construct the ApplicationClient instance using the properties file
- Construct the event that needs to be published
- Specify the event name, Device Type, Device ID and publish the event using publishEventOverHTTP() method as follows,

```
ApplicationClient myClient = new ApplicationClient(props);

JsonObject event = new JsonObject();
event.addProperty("name", "foo");
event.addProperty("cpu",  90);
```

```
event.addProperty("mem",  70);

code = myClient.publishEventOverHTTP(deviceType, deviceId, "blink", event);
```

The complete code can be found in the application example HttpApplicationDeviceEventPublish

Based on the settings in the properties file, the publishEventOverHTTP() method either publishes the event in Quickstart or in Registered flow. When the Organization ID mentioned in the properties file is quickstart, publishEventOverHTTP() method publishes the event to IoT Platform quickstart service and publishes the event in plain HTTP format. But when valid registered organization is mentioned in the properties file, this method always publishes the event in HTTPS (HTTP over SSL), so all the communication is secured.

The event in HTTP(s) is published at most once Quality of Service, so the application needs to implement the retry logic when there is an error.

10.8 Publishing commands to devices

Applications can publish commands to connected devices.

```
myClient.connect()

//Generate the event to be published
JsonObject data = new JsonObject();
data.addProperty("name", "stop-rotation");
data.addProperty("delay",  0);

//Registered flow allows 0, 1 and 2 QoS
myAppClient.publishCommand(deviceType, deviceId, "stop", data);
```

10.9 Examples

- MQTTApplicationDeviceEventPublish - A sample application that shows how to publish device events.
- RegisteredApplicationCommandPublish - A sample application that shows how to publish a command to a device.
- RegisteredApplicationSubscribeSample - A sample application that shows how to subscribe for various events like, device events, device commands, device status and application status.
- SharedSubscriptionSample - A sample application that shows how to build a scalable application which will load balance messages across multiple instances of the application.

CHAPTER 11

C# for Application Developers

- See iot-csharp in GitHub

11.1 Constructor

The constructor builds the client instance, and accepts arguments containing the following definitions:

- `orgId` - Your organization ID.
- `appId` - The unique ID of your application within your organization.
- `auth-key` - API key (required if auth-method is `apikey`).
- `auth-token` - API key token (required if auth-method is `apikey`).

If only `appId` is provided, the client will connect to the IoT Platform Quickstart service and default to an unregistered device. The argument lists create definitions which are used to interact with the IoT Platform module.

```
ApplciationClient applicationClient = new ApplciationClient(orgId, appId, apiKey, authToken);
applicationClient.connect();
```

11.2 Subscribing to device events

Events are the mechanism by which devices publish data to the IoT Platform. 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.

By default, applications will subscribe to all events from all connected devices. Use the type, id, event and msgFormat parameters to control the scope of the subscription. A single client can support multiple subscriptions. The code samples below give examples of how to subscribe to devices dependent on device type, id, event and msgFormat parameters.

11.2.1 To subscribe to all events from all devices

```
applicationClient.connect();
applicationClient.subscribeToDeviceEvents();
```

11.2.2 To subscribe to all events from all devices of a specific type

```
applicationClient.connect();
applicationClient.subscribeToDeviceEvents(deviceType);
```

11.2.3 To subscribe to a specific event from all devices

```
applicationClient.connect();
applicationClient.subscribeToDeviceEvents(evt);
```

11.2.4 To subscribe to a specific event from two or more different devices

```
 applicationClient.connect();
applicationClient.subscribeToDeviceEvents(deviceType, deviceId, evt);
```

11.2.5 To subscribe to all events published by a device in json format

```
applicationClient.connect();
applicationClient.subscribeToDeviceEvents(deviceType, deviceId, evt, "json", 0);
```

11.2.6 Handling events from devices

To process the events received by your subscriptions you need to register an event callback method.

- `event.device` - string (uniquely identifies the device across all types of devices in the organization)
- `eventName` - string
- `eventFormat` - string
- `eventData` - string

```
public static void processEvent(String eventName, string format, string data) {
    // Do something
}

applicationClient.connect();
applicationClient.eventCallback += processEvent;
applicationClient.subscribeToDeviceEvents();
```

11.3 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.

11.3.1 Subscribe to status updates for all devices

```
applicationClient.connect();
applicationClient.subscribeToDeviceStatus += processDeviceStatus;
applicationClient.subscribeToDeviceStatus();
```

11.3.2 Subscribe to status updates for two different devices

```
applicationClient.connect();
applicationClient.subscribeToDeviceStatus += processDeviceStatus;
applicationClient.subscribeToDeviceStatus(deviceType, deviceId);
```

11.3.3 Handling status updates from devices

To process the status updates received by your subscriptions you need to register an event callback method.

```
public static void processDeviceStatus(String deviceType, string deviceId, string data)
    {
       //
    }


applicationClient.connect();

applicationClient.appStatusCallback += processAppStatus;
applicationClient.subscribeToApplicationStatus();
```

11.4 Publishing events from devices

Applications can publish events as if they originated from a Device.

```
applicationClient.connect();
applicationClient.publishEvent(deviceType, deviceId, evt, data, 0);
```

11.5 Publishing commands to devices

Applications can publish commands to connected devices.

```
applicationClient.connect();
applicationClient.publishCommand(deviceType, deviceId, "testcmd", "json", data, 0);
```

CHAPTER 12

HTTP API for Devices

Important: This feature is currently available as part of a limited beta. Future updates may include changes incompatible with the current version of this feature. Try it out and let us know what you think

12.1 Publish an event

As an alternative to MQTT it is possible for devices to submit events to the IoT Platform over an HTTP API. Devices may submit a `POST` request to: `https://${orgid}.internetofthings.ibmcloud.com>/api/v0002/device/types/${typeId}/devices/$`

For full details, see the relevant API documentation. The request body (event payload) can have any content, however, MQTT message payload guidelines should be followed.

Note: When connecting a device or application to Quickstart you should use the following URL instead: `http://quickstart.internetofthings.ibmcloud.com>/api/v0002/device/types/${typeId}/devices/$`

Important: HTTP messaging only covers the ability to submit device events currently. Device management and command control require MQTT connectivity in your device.

12.1.1 Authentication

Requests must include an authorization header. Basic authentication the only method supported. Applications are authenticated by API keys, when an application makes any request to the IoT Platform API it must present an API as it's credentials:

- username = "use-token-auth"
- password = Authentication token

12.1.2 Content-Type

A `Content-Type` request header must be provided with the request. The following table shows the supported types and how they are mapped to the IoT Platform internal formats.

Content-Type Header	IoT Platform Format
text/plain	text
application/json	json
application/xml	xml
application/octet-stream	bin

12.1.3 Quality of Service

The HTTP(S) protocol provides "at most once" best effort delivery, analagous to the QoS0 quality of service provided by the MQTT protocol. When using QoS0, or the HTTP(S) equivalent, to deliver event messages, the device or application must implement retry logic to guarantee delivery.

For more information on the MQTT protocol and Quality of Service levels, please see MQTT for the IoT Platform documentation.

CHAPTER 13

MQTT Connectivity for Devices

13.1 Client connection

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

org_id.messaging.internetofthings.ibmcloud.com

13.1.1 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). We recommend the use of an encrypted connection whenever possible.

13.1.2 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 Platform 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)*.

Note: Device Support in Quickstart

When connecting to the Quickstart service no authentication (or registration) is required, and `orgId` must be set to `quickstart`

The Quickstart service does not currently support MQTT Quality of Service (QoS) levels greater than 0. This is the fastest and offers no confirmation of receipt. If you are writing device code for use with Quickstart you must also 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

Also, messages sent from devices at a rate greater than 1 per second may be discarded.

13.2 MQTT client identifier

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

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.
- **type_id** is intended to be used as an identifier 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.

Note: You can use any scheme of your choice when assigning values for `type_id` and `device_id`, however the following restrictions apply to both:

- Maximum length of 36 characters
- Must comprise only alpha-numeric characters (`a-z`, `A-Z`, `0-9`) and the following special characters:
  - dash (`-`)
  - underscore (_)
  - dot (`.`)

13.3 MQTT authentication

13.3.1 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.

13.3.2 Password

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

13.4 Publishing events

Devices can only publish to the event topic of the form iot-2/evt/**event_id**/fmt/**format_string**

- **event_id** is the ID of the event, for example "status". The event ID can be any string permitted by MQTT. Subscriber applications must use this string in their subscription topic to receive the events published on this topic if wildcards are not used.
- **format_string** is the format of the event payload, for example "json". The format can be any string permitted by MQTT. Subscriber applications must use this string in their subscription topic to receive events published on this topic if wildcards are not used. If the format is not "json", then messages will not be stored in the Historian.

Important: The message payload is limited to a maximum of 4096 bytes. Messages larger than this will be rejected.

13.5 Subscribing to commands

Devices can only subscribe to command topics of the form iot-2/cmd/**command_id**/fmt/**format_string**. They cannot subscribe to other devices' events and will only receive commands published specifically to the device itself.

- **command_id** is the ID of the command, for example "update". The command ID can be any string permitted by MQTT. A device must use this string in its subscription topic in order to receive commands published on this topic if wildcards are not used.
- **format_string** is the format of the command payload, for example "json". The format can be any string permitted by MQTT. A device must use this string in its subscription topic in order to receive commands published on this topic if wildcards are not used.

13.6 Managed Devices

Support for device lifecycle management is optional, the device management protocol used by IoTF utilises the same MQTT connection that your device already uses for events and command control.

13.6.1 Quality of Service Levels and Clean Session

Managed devices can publish messages with Quality of Service (QoS) level of 0 or 1. If QoS 1 is used, messages from the device will be queued if necessary. Messages from the device must not be retained messages.

The IoT Platform publishes requests with a QoS level of 1 to support queuing of messages. In order to queue messages sent while a managed device is not connected, the device should use `cleansession=false`.

Warning: If your managed device uses a durable subscription (cleansession=false) you need to be aware that device management commands sent to your device while it is offline will be reported as failed operations if the device does not reconnect to the service before the request times out, however when the device later connects those requests will be actioned by the device.
When handling request failures it is important to take this into account if you are using durable subscriptions for your managed devices.

13.6.2 Topics

A managed device is required to subscribe to two topics to handle requests and responses from IoTF:

- The managed device will subscribe to device management reponses on `iotdm-1/response/+`
- The managed device will subscribe to device management requests on `iotdm-1/+`

A managed device will publish to two topics:

- The managed device will publish device management responses on `iotdevice-1/response/`
- The managed device will publish device management requests on `iotdevice-1/`

13.6.3 Message Format

All messages are sent in JSON format. There are two types of message.

1. **Request** Requests are formatted as follows:

```
{  "d": {...}, "reqId": "b53eb43e-401c-453c-b8f5-94b73290c056" }
```

   - `d` carries any data relevant to the request
   - `reqId` is an identifier of the request, and must be copied into a response. If a response is not required, the field should be omitted.

2. **Response** Responses are formatted as follows:

```
{
    "rc": 0,
    "message": "success",
    "d": {...},
    "reqId": "b53eb43e-401c-453c-b8f5-94b73290c056"
}
```

   - "rc" is a result code of the original request.
   - `message` is an optional element with a text description of the response code.
   - `d` is an optional data element accompanying the response.
   - `reqId` is the request ID of the original request. This is used to correlate responses with requests, and the device needs to ensure that all request IDs are unique. When responding to IoT Platform requests, the correct `reqId` value must be sent in the response.

CHAPTER 14

Device Management Protocol

14.1 Introduction

The Device Management capabilities in the IoT Platform create a new class of connected devices, Managed Devices.

Managed Devices must, by definition, contain a management agent which can understand the IoT Platform Device Management Protocol, and send a Manage Device request to the IoT Platform Device Management server. Managed devices can access the device management operations as explained later in this document.

The Device Management Protocol defines a set of supported operations. A device management agent can support a subset of the operations, but the Manage device and Unmanage device operations must be supported. A device supporting firmware action operations must also support observation.

The Device Management Protocol is built on top of MQTT. For details specific to how Device Management Protocol interacts with MQTT please see MQTT Connectivity for Devices

14.1.1 The Device Management Lifecycle

1. A device and its associated device type are created in the IoT Platform using the dashboard or API.
2. The device connects to the IoT Platform and uses the 'Manage Device' operation to become a managed device.
3. The device's metadata, as described in the Device Model can now be viewed and manipulated through device operations, for example, firmware update and device reboot.
4. The device can communicate updates through the device-management protocol, such as location or diagnostic information and error codes.
5. In order to provide a way to deal with defunct devices in large device populations, the 'Manage device' operation request has an optional lifetime parameter. This lifetime parameter is the number of seconds within which the device must make another 'Manage device' request in order to avoid being marked as dormant and becoming an unmanaged device.
6. When a device is decommissioned it can be removed from the IoT Platform using the dashboard or REST API.

14.1.2 Return Code Summary

There are several return codes which are sent in response to the actions listed above.

- 200: Operation succeeded
- 202: Accepted (for initiating commands)

- 204: Changed (for attribute updates)
- 400: Bad request, for example, if a device is not in the appropriate state for this command
- 404: Attribute was not found, this code is also used if the operation was published to an invalid topic.
- 409: Resource could not be updated due to a conflict, for example, the resource is being updated by two simultaneous requests, so update could be retried later
- 500: Unexpected device error
- 501: Operation not implemented

14.2 Manage Device

A device uses this request to become a managed device. It should be the first device management request sent by the device after connecting to the IoT Platform. It would be usual for a device management agent to send this whenever it starts or restarts.

Important: Support for this operation is mandatory for any managed devices.

14.2.1 Topic

```
iotdevice-1/mgmt/manage
```

14.2.2 Message Format

For the request, the `d` field and all of its sub-fields are optional. The `metadata` and `deviceInfo` field values replace the corresponding attributes for the sending device if they are sent.

The optional `lifetime` field specifies the length of time in seconds within which the device must send another 'Manage device' request in order to avoid being reverted to an unmanaged device and marked as dormant. If omitted or if set to `0`, the managed device will not become dormant. When set, the minimum supported setting is `3600` (1 hour).

Optional `supports.deviceActions` and `supports.firmwareActions` indicate the capabilities of the device management agent. If `supports.deviceActions` is set, the agent supports Reboot and Factory Reset actions. For a device that does not distinguish between rebooting and factory reset, it is acceptable to use the same behaviour for both actions. If `supports.firmwareActions` is set, the agent supports Firmware Download and Firmware Update actions.

Request Format:

```
{
        "d": {
                "metadata":{},
                "lifetime": number,
                "supports": {
                        "deviceActions": boolean,
                        "firmwareActions": boolean
                },
                "deviceInfo": {
```

```
                        "serialNumber": "string",
                        "manufacturer": "string",
                        "model": "string",
                        "deviceClass": "string",
                        "description" :"string",
                        "fwVersion": "string",
                        "hwVersion": "string",
                        "descriptiveLocation": "string"
                }
        },
        "reqId": "string"
}
```

Response Format:

```
{
        "rc": 200,
        "reqId": "string"
}
```

14.2.3 Response Codes

- 200: The operation was successful.
- 400: The input message does not match the expected format, or one of the values is out of the valid range.
- 404: The topic name is incorrect, or the device is not in the database.
- 409: A conflict occurred during the device database update. To resolve this, simplify the operation is necessary.

14.3 Unmanage Device

A device uses this request when it no longer needs to be managed. The IoT Platform will no longer send new device management requests to this device and all device management requests from this device will be rejected other than a 'Manage device' request.

Important: Support for this operation is mandatory for any managed devices.

14.3.1 Topic

```
iotdevice-1/mgmt/unmanage
```

14.3.2 Message Format

Request Format:

```
{
        "reqId": "string"
}
```

Response Format:

```
{
        "rc": 200,
        "reqId": "string"
}
```

14.3.3 Response Codes

- 200: The operation was successful.
- 400: The input message does not match the expected format, or one of the values is out of the valid range.
- 404: The topic name is incorrect, or the device is not in the database.
- 409: A conflict occurred during the device database update. To resolve this, simplify the operation is necessary.

14.4 Update Location

Devices can change their location over time. The update of the location can happen in two ways:

- The device itself notifies the IoT Platform about the location update: The device retrieves its location from a GPS receiver and sends a device management message to the IoT Platform to update its location. The timestamp captures the time at which the location was retrieved from the GPS receiver. This means that the timestamp is valid, even if the transmission of the location message was delayed. In the event that the timestamp is omitted from the device management message sent, the current date and time on message receipt will be used when the device's location metadata is updated.
- A user / app updates the location of a device using the Rest API: The IoT Platform REST API is used to set the location metadata of a static device. This can be done at the time that the device is registered, or later if required. It is optional whether to include a timestamp. If omitted, the current date and time will be set as the device's location metadata.

14.4.1 Location update triggered by device

Devices that can determine their location can choose to notify the IoT Platform device management server about location changes.

14.4.2 Topic

```
iotdevice-1/device/update/location
```

14.4.3 Location update triggered by user or app

When a user or application updates the location of an active managed device the device retrieves an update message:

14.4.4 Topic

```
iotdm-1/device/update
```

14.4.5 Message Format

The "measuredDateTime" is the date of location measurement. The "updatedDateTime" is the date of the update to the device information. For efficiency reasons, the IoT Platform may batch updates to location information so the updates may be slightly delayed. The "latitude" and "longitude" should be specified in decimal degrees using WGS84.

Whenever location is updated, the values provided for latitude, longitude, elevation and uncertainty are considered as a single multi-value update. The latitude and longitude are mandatory and must both be provided with each update. Elevation and uncertainty are optional and can be omitted.

If an optional value is provided on an update and then omitted on a later update, the earlier value is deleted by the later update. Each update is considered as a complete multi-value set.

14.4.6 Location update triggered by device

Request Format:

```
{
        "d": {
                "longitude": number,
                "latitude": number,

                "elevation": number,
                "measuredDateTime": "string in ISO8601 format",
                "updatedDateTime": "string in ISO8601 format",
                "accuracy": number
        },
        "reqId": "string"
}
```

Response Format:

```
{
        "rc": 200,
        "reqId": "string"
}
```

14.4.7 Response Codes

- 200: The operation was successful.
- 400: The input message does not match the expected format, or one of the values is out of the valid range.
- 404: The topic name is incorrect, or the device is not in the database.
- 409: A conflict occurred during the device database update. To resolve this, simplify the operation is necessary.

14.4.8 Location update triggered by user or app

Payload Format:

```
{
    "d": {
        "fields": [
            {
                "field": "location",
                "value": {
                    "latitude": number,
                    "longitude": number,
                    "elevation": number,
                    "accuracy": number,
                    "measuredDateTime": "string in ISO8601 format"
                }
            }
        ]
    }
}
```

Please note: there is no reqId as no response by device is required.

14.5 Update Device Attributes

The IoT Platform can send this request to a device to update values of one or more device attributes. Attributes that can be updated by the Rest API are location, metadata, device information and firmware.

The "value" is the new value of the device attribute. It is a complex field matching the device model. Only writeable fields should be updated as a result of this operation. Values can be updated in:

- location (see Update location section for details)
- metadata (Optional)
- deviceInfo (Optional)
- mgmt.firmware (see Firmware update process for details)

14.5.1 Topic

```
iotdm-1/device/update
```

14.5.2 Message format

Payload Format:

```
{
        "d": {
                "fields": [
                        {
                                "field": "location",
                                "value": ""
                        }
                ]
        }
}
```

14.6 Add Error Code

Devices can choose to notify the IoT Platform device management server about changes in their error status.

14.6.1 Topic

```
iotdevice-1/add/diag/errorCodes
```

14.6.2 Message Format

The "errorCode" is a current device error code that needs to be added to the IoT Platform.

Request Format:

```
{
        "d": {
                "errorCode": number
        },
        "reqId": "string"
}
```

Response Format:

```
{
        "rc": 200,
        "reqId": "string"
}
```

14.6.3 Response Codes

- 200: The operation was successful.
- 400: The input message does not match the expected format, or one of the values is out of the valid range.
- 404: The topic name is incorrect, or the device is not in the database.
- 409: A conflict occurred during the device database update. To resolve this, simplify the operation is necessary.

14.7 Clear Error Codes

Devices can request that the Internet of Things Foundation clear all of their error codes.

14.7.1 Topic

```
iotdevice-1/clear/diag/errorCodes
```

14.7.2 Message Format

Request Format:

```
{
        "reqId": "string"
}
```

Response Format:

```
{
        "rc": 200,
        "reqId": "string"
}
```

14.7.3 Response Codes

- 200: The operation was successful.
- 400: The input message does not match the expected format, or one of the values is out of the valid range.
- 404: The topic name is incorrect, or the device is not in the database.
- 409: A conflict occurred during the device database update. To resolve this, simplify the operation is necessary.

14.8 Add Log

Devices can choose to notify IoTF device management support about changes a new log entry. Log entry includes a log messages, its timestamp and severity, as well as an optional base64-encoded binary diagnostic data.

14.8.1 Topic

```
iotdevice-1/add/diag/log
```

14.8.2 Message Format

“message” is a diagnostic message that needs to be added to IoTF. “timestamp” is a date and time of the log entry in ISO8601 format. “data” is an optional base64-encoded diagnostic data. “severity” is a severity of the message (0: informational, 1: warning, 2: error).

Request Format:

```
{
        "d": {
                "message": string,
                "timestamp": string,
                "data": string,
                "severity": number
```

```
        },
        "reqId": "string"
}
```

Response Format:

```
{
        "rc": 200,
        "reqId": "string"
}
```

14.8.3 Response Codes

- 200: The operation was successful.
- 400: The input message does not match the expected format, or one of the values is out of the valid range.
- 404: The topic name is incorrect, or the device is not in the database.
- 409: A conflict occurred during the device database update. To resolve this, simplify the operation is necessary.

14.9 Clear Logs

Devices can request that the Internet of Things Foundation clear all of their log entries.

14.9.1 Topic

```
iotdevice-1/clear/diag/log
```

14.9.2 Message format

Request Format:

```
{
        "reqId": "string"
}
```

Response Format:

```
{
        "rc": 200,
        "reqId": "string"
}
```

14.9.3 Response Codes

- 200: The operation was successful.
- 400: The input message does not match the expected format, or one of the values is out of the valid range.
- 404: The topic name is incorrect, or the device is not in the database.

- 409: A conflict occurred during the device database update. To resolve this, simplify the operation is necessary.

14.10 Observe Attribute Changes

The IoT Platform can send this request to a device to observe changes of one or more device attributes. When the device receives this request, it must send a notification request ("notify" message) to the IoT Platform whenever the observed attributes value changes.

Important: Devices must implement observe, notify & cancel operations in order to support *Firmware Actions - Update*.

14.10.1 Topic

```
iotdm-1/observe
```

14.10.2 Message format

The "fields" field is an array of the device attribute names from the device model. For example, values could be "location", "mgmt.firmware" or "mgmt.firmware.state". If a complex field, such as "mgmt.firmware" is specified, it is expected that its underlying fields are updated at the same time, such that only a single notify message is generated.

The "message" field used in the response can be specified if "rc" is not 200. If any field value which was to be observed could not be retrieved, "rc" should be set to 404 (if not found) or 500 (any other reason). When values for fields to be observed cannot be found, "fields" should contain an array of elements with "field" set to the name of each field that could not be read, "value" fields should be omitted. For the response code to be set to 200, both "field" and "value" must be specified, "value" is the current value of an attribute identified by "field" content.

Request Format:

```
{
        "d": {
                "fields": [
                        "string"
                ]
        },
        "reqId": "string"
}
```

Response Format:

```
{
        "rc": number,
        "message": "string",
        "d": {
                "fields": [
                        {
                                "field": "field_name",
                                "value": "field_value"
                        }
                ]
```

```
        },
        "reqId": "string"
}
```

14.11 Cancel Attribute Observation

The IoT Platform can send this request to a device to cancel the current observation of one or more device attributes. The "fields" is an array of the device attribute names from the device model, for example, values could be "location", "mgmt.firmware" or "mgmt.firmware.state".

The "message" field must be specified if "rc" is not 200.

Important: Devices must implement observe, notify & cancel operations in order to support *Firmware Actions - Update*.

14.11.1 Topic

```
iotdm-1/cancel
```

14.11.2 Message format

Request Format:

```
{
        "d": {
                "fields": [
                        "string"
                ]
        },
        "reqId": "string"
}
```

Response Format:

```
{
        "rc": number,
        "message": "string",
        "reqId": "string"
}
```

14.12 Notify Attribute Changes

The IoT Platform can make an observation request referring to a specific attribute or set of values. When the value of the attribute or attributes changes, the device must send a notification containing the latest value.

The "field_name" value is the name of the attribute that has changed, the "field_value" is the current value of the attribute. The attribute can be a complex field, if multiple values in a complex field are updated as a result of a single operation, only a single notification message should be sent.

If notify request is processed successfully, "rc" should be set to 200. If the request is not correct, "rc" should be set to 400. If the field specified in the notify request is not being observed, "rc" should be set to 404.

Important: Devices must implement observe, notify & cancel operations in order to support *Firmware Actions - Update*.

14.12.1 Topic

```
iotdevice-1/notify
```

14.12.2 Message format

Request Format:

```
{
        "d": {
                "field": "field_name",
                "value": "field_value"
        }
        "reqId": "string"
}
```

Response Format:

```
{
        "rc": number,
        "reqId": "string"
}
```

14.12.3 Response Codes

- 200: The operation was successful.
- 400: The input message does not match the expected format, or one of the values is out of the valid range.
- 404: The topic name is incorrect, the device is not in the database, or there is no observation for the field reported.
- 409: A conflict occurred during the device database update. To resolve this, simplify the operation is necessary.
- 500: An internal error occurred, contact IBM Support.

CHAPTER 15

Device Management Requests

15.1 Device Actions - Reboot

The IoT Platform can send this request to reboot a device. The action is considered complete when the device sends a Manage device request following its reboot.

If this operation can be initiated immediately, set “rc” to 202, if reboot attempt fails, the “rc” is set to 500 and the “message” field should be set accordingly, if the reboot is not supported, set “rc” to 501 and optionally set “message” accordingly.

15.1.1 Topic

```
iotdm-1/mgmt/initiate/device/reboot
```

15.1.2 Message format

Request Format:

```
{
        "reqId": "string"
}
```

Response Format:

```
{
        "rc": "response_code",
        "message": "string",
        "reqId": "string"
}
```

15.2 Device Actions - Factory Reset

The IoT Platform can send this request to reset the device to factory settings, as part of this process, the device also reboots. The action is considered complete when the device sends a Manage device request following its reboot.

The response code should be 202 if this action can be initiated immediately. If the factory reset attempt fails, the "rc" should be 500 and the "message" field should be set accordingly, if the factory reset action is not supported, set "rc" to 501 and optionally set "message" accordingly.

15.2.1 Topic

```
iotdm-1/mgmt/initiate/device/factory_reset
```

15.2.2 Message format

Request Format:

```
{
        "reqId": "string"
}
```

Response Format:

```
{
        "rc": "response_code",
        "message": "string",
        "reqId": "string"
}
```

15.3 Firmware Actions

The firmware level currently known to be on a given device is stored in the `deviceInfo.fwVersion` attribute. The `mgmt.firmware` attributes are used to perform a firmware update and observe its status.

Important: The managed device must support observation of the `mgmt.firmware` attribute in order to support firmware actions.

The firmware update process is separated into two distinct actions, Downloading Firmware, and Updating Firmware. The status of each of these actions is stored in a separate attribute on the device. The `mgmt.firmware.state` attribute describes the status of the firmware download. The possible values for `mgmt.firmware.state` are:

Value	State	Meaning
0	Idle	The device is currently not in the process of downloading firmware
1	Downloading	The device is currently downloading firmware
2	Downloaded	The device has successfully downloaded a firmware update and it is ready to install

The `mgmt.firmware.updateStatus` attribute describes the status of firmware update. The possible values for `mgmt.firmware.updateStatus` are:

Value	State	Meaning
0	Success	The firmware has been successfully updated
1	In Progress	The firmware update has been initiated but is not yet complete
2	Out of Memory	An out of memory condition has been detected during the operation.
3	Connection Lost	The connection was lost during the firmware download
4	Verification Failed	The firmware did not pass verification
5	Unsupported Image	The downloaded firmware image is not supported by the device
6	Invalid URI	The device could not download the firmware from the provided URI

15.4 Firmware Actions - Download

The Download Firmware action can be initiated by using either the IoT Platform dashboard, or the REST API.

To initiate a firmware download using the REST API, issue a POST request to /mgmt/requests. The information provided is:

- The action `firmware/download`
- The URI for the firmware image
- A list of devices to receive the image, with a maximum of 5000 devices
- Optional verifier string to validate the image
- Optional firmware name
- Optional firmware version

Example firmware download request on which all the following example messages are based:

```
{
   "action" : "firmware/download",
   "parameters" : [{
         "name" : "uri",
         "value" : "some uri for firmware location"
      }, {
         "name" : "name",
         "value" : "some firmware name"
      }, {
         "name" : "verifier",
         "value" : "some validation code"
      }, {
         "name" : "version",
         "value" : "some firmware version"
      }
   ],
   "devices" : [{
         "typeId" : "someType",
         "deviceId" : "someId"
      }
   ]
}
```

The device management server in the IoT Platform uses the Device Management Protocol to send a request to the devices, initiating the firmware download. There are multiple steps:

1. Firmware details update request sent on topic `iotdm-1/device/update`:

   This request let the device validate if the requested firmware differs from the currently installed firmware. If there is a difference, set `rc` to `204`, which translates to the status `Changed`. The following example shows which message is to be expected for the previously sent example firmware download request and what response should be sent, when a difference is detected:

```
Incoming request from the IoT Platform:

Topic: iotdm-1/device/update
Message:
{
   "reqId" : "f38faafc-53de-47a8-a940-e697552c3194",
   "d" : {
      "fields" : [{
            "field" : "mgmt.firmware",
            "value" : {
               "version" : "some firmware version",
               "name" : "some firmware name",
               "uri" : "some uri for firmware location",
               "verifier" : "some validation code",
               "state" : 0,
               "updateStatus" : 0,
               "updatedDateTime" : ""
            }
         }
      ]
   }
}

Outgoing response from device:

Topic: iotdevice-1/response
Message:
{
   "rc" : 204,
   "reqId" : "f38faafc-53de-47a8-a940-e697552c3194"
}
```

This response will trigger the next request.

2. Observation request for firmware download status `iotdm-1/observe`:

   Verifies if the device is ready to start the firmware download. When the download can be started immediately, set `rc` to `200` (`Ok`), `mgmt.firmware.state` to `0` (`Idle`) and `mgmt.firmware.updateStatus` to `0` (`Idle`). Here an example exchange between the IoT Platform and device:

```
Incoming request from the IoT Platform:

Topic: iotdm-1/observe
Message:
{
   "reqId" : "909b477c-cd37-4bee-83fa-1d568664fbe8",
   "d" : {
      "fields" : [ {
```

```
              "fields" : "mgmt.firmware"
              }
      ]
   }
}

Outgoing response from device:

Topic: iotdevice-1/response
Message:
{
   "rc" : 200,
   "reqId" : "909b477c-cd37-4bee-83fa-1d568664fbe8"
}
```

This exchange will trigger the last step.

3. Initiate the download request sent on topic `iotdm-1/mgmt/initiate/firmware/download`:

   This request tells a device to actually start the firmware download. If the action can be initiated immediately, set `rc` to `202`. Here an example:

```
Incoming request from the IoT Platform:

Topic: iotdm-1/mgmt/initiate/firmware/download
Message:
{
   "reqId" : "7b244053-c08e-4d89-9ed6-6eb2618a8734"
}

Outgoing response from device:

Topic: iotdevice-1/response
Message:
{
   "rc" : 202,
   "reqId" : "7b244053-c08e-4d89-9ed6-6eb2618a8734"
}
```

After a firmware download is initiated this way, the device needs to report to the IoT Platform the status of the download. This is possible by publishing a message to the `iotdevice-1/notify`-topic, where the `mgmt.firmware.state` is set to either `1` (`Downloading`) or `2` (`Downloaded`). Here some examples:

```
Outgoing message from device:

Topic: iotdevice-1/notify
Message:
{
   "reqId" : "123456789";
   "d" : {
```

```
      "fields" : [ {
             "fields" : "mgmt.firmware",
             "value" : {
                    "state" : 1
             }
      } ]
   }
}


Wait some time...


Outgoing message from device:

Topic: iotdevice-1/notify
Message:
{
   "reqId" : "1234567890";
   "d" : {
      "fields" : [ {
             "fields" : "mgmt.firmware",
             "value" : {
                    "state" : 2
             }
      } ]
   }
}
```

After the notification with `mgmt.firmware.state` set to `2` was published, a request will be triggered on the `iotdm-1/cancel`-topic, which cancels the observation of the `mgmt.firmware`-field. After a response with `rc` set to `200` was sent the firmware download is completed. Example:

```
Incoming request from the IoT Platform:

Topic: iotdm-1/cancel
Message:
{
   "reqId" : "d9ca3635-64d5-46e2-93ee-7d1b573fb20f",
   "d" : {
      "fields" : [{
            "field" : "mgmt.firmware"
         }
      ]
   }
}


Outgoing message from device:

Topic: iotdevice-1/response
Message:
{
```

```
   "rc" : 200,
   "reqId" : "d9ca3635-64d5-46e2-93ee-7d1b573fb20f"
}
```

Useful information regarding error handling:

- If `mgmt.firmware.state` is not `0` ("Idle") an error should be reported with response code `400`, and an optional message text.-
- If `mgmt.firmware.uri` is not set or is not a valid URI, set `rc` to `400`.
- If firmware download attempt fails, set `rc` to `500` and optionally set `message` accordingly.
- If firmware download is not supported, set `rc` to `501` and optionally set `message` accordingly.
- When an excecute request is received by the device, `mgmt.firmware.state` should change from `0` (Idle) to `1` (Downloading).
- When the download has been completed successfully, `mgmt.firmware.state` should be set to `2` (Downloaded).
- If an error occurrs during download `mgmt.firmware.state` should be set to `0` (Idle) and `mgmt.firmware.updateStatus` should be set to one of the error status values:
  - 2 (Out of Memory)
  - 3 (Connection Lost)
  - 6 (Invalid URI)
- If a firmware verifier has been set, the device should attempt to verify the firmware image. If the image verification fails, `mgmt.firmware.state` should be set to `0` (Idle) and `mgmt.firmware.updateStatus` should be set to the error status value `4` (Verification Failed).

15.5 Firmware Actions - Update

The installation of the downloaded firmware is initiated using the REST API by issuing a POST request to /mgmt/requests. The information which should be provided is:

- The action `firmware/update`
- The list of devices to receive the image, all of the same device type.

Here an example request:

```
{
   "action" : "firmware/update",
   "devices" : [{
         "typeId" : "someType",
         "deviceId" : "someId"
      }
   ]
}
```

In order to monitor the status of the firmware update the IoT Platform first triggers an observer request on the topic `iotdm-1/observe`. When the device is ready to start the update process it sents a response with `rc` set to `200`, `mgmt.firmware.state` set to `0` and `mgmt.firmware.updateStatus` set to `0`. Here an example:

```
Incoming request from the IoT Platform:

Topic: iotdm-1/observe
Message:
{
   "reqId" : "909b477c-cd37-4bee-83fa-1d568664fbe8",
   "d" : {
      "fields" : ["mgmt.firmware"]
   }
}

Outgoing response from device:

Topic: iotdevice-1/response
Message:
{
   "rc" : 200,
   "reqId" : "909b477c-cd37-4bee-83fa-1d568664fbe8",
   "d" : {
      "fields" : [{
            "field" : "mgmt.firmware",
            "value" : {
               "state" : 0,
               "updateStatus" : 0
            }
         }
      ]
   }
}
```

Afterwards the device management server in the IoT Platform uses the device management protocol to request that the devices specified initiate the firmware installation by publishing using the topic `iotdm-1/mgmt/initiate/firmware/update`. If this operation can be initiated immediately, `rc` should be set to `202`. If firmware was not previously downloaded successfully, `rc` should be set to `400`. Here some example exchange:

```
Incoming request from the IoT Platform:

Topic: iotdm-1/mgmt/initiate/firmware/update
Message:
{
   "reqId" : "7b244053-c08e-4d89-9ed6-6eb2618a8734"
}

Outgoing response from device:
```

```
Topic: iotdevice-1/response
Message:
{
   "rc" : 202,
   "reqId" : "7b244053-c08e-4d89-9ed6-6eb2618a8734"
}
```

In order to finish the firmware update request the device has to report its update status to the IoT Platform via a status message published on its `iotdevice-1/notify`-topic. Once firmware update is completed, `mgmt.firmware.updateStatus` should be set to `0` (`Success`), `mgmt.firmware.state` should be set to `0` (`Idle`), downloaded firmware image can be deleted from the device and `deviceInfo.fwVersion` should be set to the value of `mgmt.firmware.version`. Here an example notify message:

```
Outgoing message from device:

Topic: iotdevice-1/notify
Message:
{
   "d" : {
      "field" : "mgmt.firmware",
      "value" : {
         "state" : 0,
         "updateStatus" : 0
      }
   }
}
```

After the IoT Platform received the notify of a completed firmware update it will trigger a last request on the `iotdm-1/cancel`-topic for cancelation of the observation of the `mgmt.firmware`-field. After a response with `rc` set to `200` was sent the firmware update request is completed. Example:

```
Incoming request from the IoT Platform:

Topic: iotdm-1/cancel
Message:
{
   "reqId" : "d9ca3635-64d5-46e2-93ee-7d1b573fb20f",
   "d" : {
      "data" : [{
            "field" : "mgmt.firmware"
         }
      ]
   }
}


Outgoing message from device:
```

```
Topic: iotdevice-1/response
Message:
{
   "rc" : 200,
   "reqId" : "d9ca3635-64d5-46e2-93ee-7d1b573fb20f"
}
```

Useful information regarding error and process handling:

- If firmware update attempt fails, `rc` should be set to `500` and the `message` field can optionally be set to contain relevant information.
- If firmware update is not supported `rc` should be set to `501` and the `message` field can optionally be set to contain relevant information.
- If `mgmt.firmware.state` is not `2` (Downloaded), an error should be reported with `rc` set to `400` and an optional message text.
- Otherwise, `mgmt.firmware.updateStatus` should be set to `1` (In Progress) and firmware installation should start.
- If firmware installation fails, `mgmt.firmware.updateStatus` should be set to either:
  - `2` (Out of Memory)
  - `5` (Unsupported Image)

Important: All fields under `mgmt.firmware` must be set at the same time, so that if there is a current observation for `mgmt.firmware`, only a single notify message is sent.

CHAPTER 16

Python for Device Developers

- See iot-python in GitHub
- See ibmiotf on PyPi

16.1 Constructor

The constructor builds the client instance, and accepts an options dict containing the following definitions:

- `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`).

If no options dict is provided, the client will connect to the IoT Platform Quickstart, and default to an unregistered device. The options dict creates definitions which are used to interact with the IoT Platform module.

```
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:
  ...
```

16.1.1 Using a configuration file

Instead of including an options dict directly, you can use a configuration file containing an options dict. If you are using a configuration file containing an options dict, use the following code format.

```
import ibmiotf.device
try:
  options = ibmiotf.device.ParseConfigFile(configFilePath)
```

```
  client = ibmiotf.device.Client(options)
except ibmiotf.ConnectionException  as e:
  ...
```

The content of the configuration file must be in the following format(need not contain $ sign):

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

16.2 Publishing events

Events are the mechanism by which devices publish data to the IoT Platform. 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 Platform 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.

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.

16.2.1 Publish event using default quality of service

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

16.2.2 Publish event using user-defined quality of service

Events can be published at higher MQTT quality of servive levels, but these events may take slower then QoS level 0, because of the extra confirmation of receipt.

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

16.3 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:

- command - string

- format - string
- data - dict
- timestamp - datetime

```
def myCommandCallback(cmd):
  print("Command received: %s" % cmd.data)
  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
```

16.4 Custom message format support

By default the library supports the encoding and decoding of Python dictionary objects as JSON when you use msgFormat "json". When you use msgFormat "json-iotf" it will encode the message in accordance with the IoT Platform JSON Payload Specification. To add support for your own custom message formats see the sample in GitHub

Once you have created your custom encoder module it needs to be registered in the device client. If you attempt to use an unknown message format when sending an event or the device receives a command send in a format that it does not know how to decode then the library will throw a MissingMessageDecoderException.

```
import myCustomCodec

client.setMessageEncoderModule("custom", myCustomCodec)
client.publishEvent("status", "custom", myData)
```

CHAPTER 17

Node.js for Device Developers

- See iot-nodejs in GitHub
- See the samples for device in Github
- See ibmiotf on NPM

17.1 Constructor

The constructor builds the device client instance. It accepts a configuration json containing the following definitions:

- `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`)

If you want to use Quickstart, then send only the first three properties.

```
var iotf = require("ibmiotf");
var config = {
            "org" : "organization",
            "id" : "deviceId",
            "type" : "deviceType",
            "auth-method" : "token",
            "auth-token" : "authToken"
};


var deviceClient = new iotf.IotfDevice(config);
```

17.1.1 Using a configuration file

Instead of passing the configuration json directly, you can also use a configuration json file. Use the following code snippet:

```
var iotf = require("ibmiotf");
var config = require("./device.json");
var deviceClient = new iotf.IotfDevice(config);
```

The configuration file *device.json* must be in the format of

```
{
  "org": "xxxxx",
  "type": "raspi",
  "id": "pi1",
  "auth-method" : "token",
  "auth-token" : "xxxxxxxxxxxxxxxx"
}
```

17.2 Connecting to the IoT Platform

Connect to the IoT Platform by calling the *connect* function.

```
var iotf = require("ibmiotf");
var config = require("./device.json");

var deviceClient = new iotf.IotfDevice(config);

//setting the log level to debug. By default its 'warn'
deviceClient.log.setLevel('debug');

deviceClient.connect();

deviceClient.on('connect', function(){
        var i=0;
        console.log("connected");
        setInterval(function function_name () {
                i++;
                deviceClient.publish('myevt', 'json', '{"value":'+i+'}', 2);
        },2000);
});
```

After the successful connection to the IoT Platform service, the device client sends a *connect* event. So all the device logic can be implemented inside this callback function.

17.3 Logging

By default, all the logs of `warn` are logged. If you want to enable more logs, use the *log.setLevel* function. Supported log levels - *trace, debug, info, warn, error*.

```
var iotf = require("ibmiotf");
var config = require("./device.json");

var deviceClient = new iotf.IotfDevice(config);

//setting the log level to debug. By default its 'warn'
deviceClient.log.setLevel('debug');
```

17.4 Publishing events

Events are the mechanism by which devices publish data to the IoT Platform. 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 Platform 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.

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. Please not that if you are using the Internet of Things Quickstart service, events can only be published at QoS level 0.

Events can be published by using:

- eventType - Type of event to be published e.g status, gps.
- eventFormat - Format of the event e.g json.
- data - Payload of the event.(Must be buffer/String)
- QoS - MQTT quality of service for the publish event. Supported values : 0,1,2.

```
  var deviceClient = new Client.IotfDevice(config);

  deviceClient.connect();

  deviceClient.on("connect", function () {
     //publishing event using the default quality of service
     deviceClient.publish("status","json",'{"d" : { "cpu" : 60, "mem" : 50 }}');

     //publishing event using the user-defined quality of service
     var myQosLevel=2
     deviceClient.publish("status","json",'{"d" : { "cpu" : 60, "mem" : 50 }}', myQosLevel);
});
```

17.5 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 function. The device client sends *command* when a command is received. The callback function has the following properties.

- commandName - name of the command invoked
- format - e.g json, xml
- payload - payload for the command
- topic - actual topic where the command was received

```
var deviceClient = new Client.IotfDevice(config);

deviceClient.connect();

deviceClient.on("connect", function () {
        //publishing event using the default quality of service
        deviceClient.publish("status","json",'{"d" : { "cpu" : 60, "mem" : 50 }}');

});

deviceClient.on("command", function (commandName,format,payload,topic) {
        if(commandName === "blink") {
                console.log(blink);
                //function to be performed for this command
                blink(payload);
```

```
        } else {
                console.log("Command not supported.. " + commandName);
        }
});
```

17.6 Handling errors

When the device clients encounters an error, it emits an *error* event.

```
var deviceClient = new Client.IotfDevice(config);

deviceClient.connect();

deviceClient.on("connect", function () {
        //publishing event using the default quality of service
        deviceClient.publish("status","json",'{"d" : { "cpu" : 60, "mem" : 50 }}');

});

deviceClient.on("error", function (err) {
        console.log("Error : "+err);
});
....
```

17.7 Disconnect Client

Disconnects the client and releases the connections

```
var deviceClient = new Client.IotfDevice(config);

deviceClient.connect();

client.on("connect", function () {
        //publishing event using the default quality of service
        client.publish("status","json",'{"d" : { "cpu" : 60, "mem" : 50 }}');

        //publishing event using the user-defined quality of service
        var myQosLevel=2
        client.publish("status","json",'{"d" : { "cpu" : 60, "mem" : 50 }}', myQosLevel);

        //disconnect the client
        client.disconnect();
});

....
```

CHAPTER 18

Java for Device Developers

- See iot-java in GitHub

18.1 Constructor

The constructor builds the client instance, and accepts a Properties object containing the following definitions:

- org - Your organization ID. (This is a required field. In case of quickstart flow, provide org as quickstart.)
- type - The type of your device. (This is a required field.)
- id - The ID of your device. (This is a required field.
- auth-method - Method of authentication (This is an optional field, needed only for registered flow and the only value currently supported is "token").
- auth-token - API key token (This is an optional field, needed only for registered flow).
- clean-session - true or false (required only if you want to connect the application in durable subscription. By default the clean-session is set to true).

Note: One must set `clean-session` to false to connect the device in durable subscription. Refer to Subscription Buffers and Clean Session for more information about the clean session.

The Properties object creates definitions which are used to interact with the IoT Platform module.

The following code shows a device publishing events in a Quickstart mode.

```
package com.ibm.iotf.sample.client.device;

import java.util.Properties;

import com.google.gson.JsonObject;
import com.ibm.iotf.client.device.DeviceClient;

public class QuickstartDeviceEventPublish {

        public static void main(String[] args) {

                //Provide the device specific data using Properties class
                Properties options = new Properties();
                options.setProperty("org", "quickstart");
```

```
                options.setProperty("type", "iotsample-arduino");
                options.setProperty("id", "00aabbccde03");

                DeviceClient myClient = null;
                try {
                        //Instantiate the class by passing the properties file
                        myClient = new DeviceClient(options);
                } catch (Exception e) {
                        e.printStackTrace();
                }

                //Connect to the IBM IoT Platform
                myClient.connect();

                //Generate a JSON object of the event to be published
                JsonObject event = new JsonObject();
                event.addProperty("name", "foo");
                event.addProperty("cpu",  90);
                event.addProperty("mem",  70);

                //Quickstart flow allows only QoS = 0
                myClient.publishEvent("status", event, 0);
                System.out.println("SUCCESSFULLY POSTED......");

...
```

The following program shows a device publishing events in a registered flow

```
package com.ibm.iotf.sample.client.device;

import java.util.Properties;

import com.google.gson.JsonObject;
import com.ibm.iotf.client.device.DeviceClient;

public class RegisteredDeviceEventPublish {

        public static void main(String[] args) {

                //Provide the device specific data, as well as Auth-key and token using Properties
                Properties options = new Properties();

                options.setProperty("org", "uguhsp");
                options.setProperty("type", "iotsample-arduino");
                options.setProperty("id", "00aabbccde03");
                options.setProperty("auth-method", "token");
                options.setProperty("auth-token", "AUTH TOKEN FOR DEVICE");

                DeviceClient myClient = null;
                try {
                        //Instantiate the class by passing the properties file
                        myClient = new DeviceClient(options);
                } catch (Exception e) {
                        e.printStackTrace();
                }

                //Connect to the IBM IoT Platform
                myClient.connect();
```

```
                //Generate a JSON object of the event to be published
                JsonObject event = new JsonObject();
                event.addProperty("name", "foo");
                event.addProperty("cpu",  90);
                event.addProperty("mem",  70);

                //Registered flow allows 0, 1 and 2 QoS
                myClient.publishEvent("status", event);
                System.out.println("SUCCESSFULLY POSTED......");

...
```

18.1.1 Using a configuration file

Instead of including a Properties object directly, you can use a configuration file containing the name-value pairs for Properties. If you are using a configuration file containing a Properties object, use the following code format.

```
 package com.ibm.iotf.sample.client.device;

 import java.io.File;
 import java.util.Properties;

 import com.google.gson.JsonObject;
 import com.ibm.iotf.client.device.DeviceClient;

 public class RegisteredDeviceEventPublishPropertiesFile {

         public static void main(String[] args) {
                //Provide the device specific data, as well as Auth-key and token using Properties
                Properties options = DeviceClient.parsePropertiesFile(new File("C:\\temp\\device.pr

                DeviceClient myClient = null;
                try {
                        //Instantiate the class by passing the properties file
                        myClient = new DeviceClient(options);
                } catch (Exception e) {
                        e.printStackTrace();
                }

                //Connect to the IBM IoT Platform
                myClient.connect();

                //Generate a JSON object of the event to be published
                JsonObject event = new JsonObject();
                event.addProperty("name", "foo");
                event.addProperty("cpu",  90);
                event.addProperty("mem",  70);

                //Registered flow allows 0, 1 and 2 QoS
                myClient.publishEvent("status", event, 1);
                System.out.println("SUCCESSFULLY POSTED......");

...
```

The content of the configuration file must be in the following format:

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

18.2 Publishing events

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

When an event is received by the IBM IoT Platform 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.

Events can be published at any of the three *quality of service levels <../messaging/mqtt.html#/>* defined by the MQTT protocol. By default events will be published as qos level 0.

18.2.1 Publish event using default quality of service

```
myClient.connect();

JsonObject event = new JsonObject();
event.addProperty("name", "foo");
event.addProperty("cpu",  90);
event.addProperty("mem",  70);

myClient.publishEvent("status", event);
```

18.2.2 Publish event using user-defined quality of service

Events can be published at higher MQTT quality of servive levels, but these events may take slower than QoS level 0, because of the extra confirmation of receipt. Also Quickstart flow allows only Qos of 0

```
myClient.connect();

JsonObject event = new JsonObject();
event.addProperty("name", "foo");
event.addProperty("cpu",  90);
event.addProperty("mem",  70);

//Registered flow allows 0, 1 and 2 QoS
myClient.publishEvent("status", event, 2);
```

18.2.3 Publish event using HTTP(s)

Apart from MQTT, the devices can publish events to the IoT Platform using HTTP(s) by following 3 simple steps,

- Construct a DeviceClient instance using the properties file
- Construct an event that needs to be published
- Specify the event name and publish the event using publishEventOverHTTP() method as follows,

```
DeviceClient myClient = new DeviceClient(deviceProps);

JsonObject event = new JsonObject();
event.addProperty("name", "foo");
event.addProperty("cpu",  90);
event.addProperty("mem",  70);

int httpCode = myClient.publishEventOverHTTP("blink", event);
```

The complete code can be found in the device example HttpDeviceEventPublish

Based on the settings in the properties file, the publishEventOverHTTP() method either publishes the event in Quickstart or in Registered flow. When the Organization ID mentioned in the properties file is quickstart, publishEventOverHTTP() method publishes the event to the IoT Platform quickstart service and publishes the event in plain HTTP format. But when valid registered organization is mentioned in the properties file, this method always publishes the event in HTTPS (HTTP over SSL), so all the communication is secured.

The event in HTTP(s) is published at most once QoS, so the device needs to implement the retry logic when there is an error.

18.3 Handling commands

When the device client connects it automatically subscribes to any commands 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 - java.lang.String
- format - java.lang.String
- command - java.lang.String
- timestamp - org.joda.time.DateTime

```
package com.ibm.iotf.sample.client.device;

import java.util.Properties;


import com.ibm.iotf.client.device.Command;
import com.ibm.iotf.client.device.CommandCallback;
import com.ibm.iotf.client.device.DeviceClient;


//Implement the CommandCallback class to provide the way in which you want the command to be handled
class MyNewCommandCallback implements CommandCallback{

        public MyNewCommandCallback() {
        }

        //In this sample, we are just displaying the command the moment the device recieves it
        @Override
```

```
        public void processCommand(Command command) {
                System.out.println("COMMAND RECEIVED = '" + command.getCommand() + "'\twith Payload =
        }
}

public class RegisteredDeviceCommandSubscribe {


        public static void main(String[] args) {

                //Provide the device specific data, as well as Auth-key and token using Properties cl
                Properties options = new Properties();

                options.setProperty("org", "uguhsp");
                options.setProperty("type", "iotsample-arduino");
                options.setProperty("id", "00aabbccde03");
                options.setProperty("auth-method", "token");
                options.setProperty("auth-token", "AUTH TOKEN FOR DEVICE");

                DeviceClient myClient = null;
                try {
                        //Instantiate the class by passing the properties file
                        myClient = new DeviceClient(options);
                } catch (Exception e) {
                        e.printStackTrace();
                }

                //Pass the above implemented CommandCallback as an argument to this device client
                myClient.setCommandCallback(new MyNewCommandCallback());

                //Connect to the IoT Platform
                myClient.connect();
        }
}
```

CHAPTER 19

C# for Device Developers

- See iot-csharp in GitHub

19.1 Constructor

The constructor builds the client instance, and accepts arguments containing the following definitions:

- `orgId` - Your organization ID.
- `deviceType` - The type of your device.
- `deviceId` - 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").

If deviceId and deviceType are provided, the client will connect to the IoT Platform Quickstart, and default to an unregistered device. The argument lists creates definitions which are used to interact with the IoT Platform module.

```
namespace com.ibm.iotf.client

public DeviceClient(string orgId, string deviceType, string deviceID, string authmethod, string autht
    : base(orgId, "d" + CLIENT_ID_DELIMITER + orgId + CLIENT_ID_DELIMITER + deviceType + CLIENT_ID_DE
{

}
```

19.2 Publishing events

Events are the mechanism by which devices publish data to the IoT Platform. 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 Platform the credentials of the connection on which the event was received are used to determine which device sent the event. With this architecture it is impossible for a device to impersonate another device.

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

19.3 Publish event using default quality of service

```
    deviceClient.connect();
deviceClient.publishEvent("event", "json", "{temp:23}");
```

19.4 Publish event using user-defined quality of service

Events can be published at higher MQTT quality of service levels, but events published at QoS levels above 0 may be slower then QoS 0 events, because of the extra confirmation of receipt used in QoS levels above 0.

```
    deviceClient.connect();
deviceClient.publishEvent("event", "json", "{temp:23}", 2);
```

19.5 Handling commands

When the device client connects, it automatically subscribes to any commands for this device. To process specific commands, you must register a command callback method.

```
      public static void processCommand(string cmdName, string format, string data) {
  ...
}
```

```
    deviceClient.connect();
deviceClient.commandCallback += processCommand;
```

CHAPTER 20

Embedded C for Device Developers

- See iotf-embeddedc on GitHub

20.1 Dependencies

- Eclipse Paho Embedded C library - provides an MQTT C client library, check here for more information.

20.2 Installation

To install the IoT Platform client library for Embedded C follow the instructions below.

1. To install the latest version of the library, enter the following code in your command line.

```
[root@localhost ~]# git clone https://github.com/ibm-messaging/iotf-embeddedc.git
```

2. Copy the Paho library .tar file that was downloaded in the previous step to the *lib* directory.

```
cd iotf-embeddedc
cp ~/org.eclipse.paho.mqtt.embedded-c-1.0.0.tar.gz lib/
```

3. Extract the library file

```
cd lib
tar xvzf org.eclipse.paho.mqtt.embedded-c-1.0.0.tar.gz
```

When downloaded, the client has the following file structure:

```
|-lib - contains all the dependent files
|-samples - contains the helloWorld and sampleDevice samples
  |-sampleDevice.c - sample device implementation
  |-helloworld.c - quickstart application
  |-README.md
  |-Makefile
  |-build.sh
|-iotfclient.c - Main client file
|-iotfclient.h - Header file for the client
```

20.3 Initializing the Client Library

After downloading the client library, it must be initialized and connected to the IoT Platform. There are 2 ways to initialize the IoT Platform Client Library for Embedded C:

20.3.1 Passing as Parameters

The 'initialize' function takes the following details to connect to the IoT Platform service:

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

```
#include "iotfclient.h"
....
....
Iotfclient client;
//quickstart
rc = initialize(&client,"quickstart","iotsample","001122334455",NULL,NULL);
//registered
rc = initialize(&client,"orgid","type","id","token","authtoken");
....
```

20.3.2 Using a Configuration File

You can also use a configuration file to initialize the Embedded C client library. The function 'initialize_configfile' takes the configuration file path as a parameter.

```
#include "iotfclient.h"
....
....
char *filePath = "./device.cfg";
Iotfclient client;
rc = initialize_configfile(&client, filePath);
....
```

The configuration file must use the following format.

```
org=$orgId
type=$myDeviceType
id=$myDeviceId
auth-method=token
auth-token=$token
```

20.4 Connecting to the Service

After initializing the IoT Platform Embedded C client library, you can connect to the IoT Platform by calling the 'connectiotf' function.

```
#include "iotfclient.h"
....
....
Iotfclient client;
char *configFilePath = "./device.cfg";

rc = initialize_configfile(&client, configFilePath);

if(rc != SUCCESS){
        printf("initialize failed and returned rc = %d.\n Quitting..", rc);
        return 0;
}

rc = connectiotf(&client);

if(rc != SUCCESS){
        printf("Connection failed and returned rc = %d.\n Quitting..", rc);
        return 0;
}
....
```

20.5 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 function by calling the function 'setCommandHandler'. The commands are returned as:

- commandName - name of the command invoked
- format - e.g json, xml
- payload

```
#include "iotfclient.h"

void myCallback (char* commandName, char* format, void* payload)
{
printf("The command received :: %s\n", commandName);
printf("format : %s\n", format);
printf("Payload is : %s\n", (char *)payload);
}
...
...
char *filePath = "./device.cfg";
rc = connectiotfConfig(filePath);
setCommandHandler(myCallback);

yield(1000);
....
```

Note: The 'yield' function must be called periodically to receive commands.

20.6 Publishing events

Events can be published by using:

- eventType - Type of event to be published e.g status, gps
- eventFormat - Format of the event e.g json
- data - Payload of the event
- QoS - qos for the publish event. Supported values : QOS0, QOS1, QOS2

```
#include "iotfclient.h"
....
rc = connectiotf (org, type, id , authmethod, authtoken);
char *payload = {\"d\" : {\"temp\" : 34 }};

rc= publishEvent("status","json", "{\"d\" : {\"temp\" : 34 }}", QOS0);
....
```

20.7 Disconnect Client

To disconnect the client and release the connections, run the following code snippet.

```
#include "iotfclient.h"
....
rc = connectiotf (org, type, id , authmethod, authtoken);
char *payload = {\"d\" : {\"temp\" : 34 }};

rc= publishEvent("status","json", payload , QOS0);
...
rc = disconnect();
....
```

20.8 Samples

Sample device and application code is provided in GitHub.

mBed C++ for Device Developers

- See ibmiotf on developer.mbed.org

The mBed C++ client library can be used to connect mBed devices like LPC1768, FRDM-K64F and etc.. to the IoT Platform Cloud service with ease. Although the library uses C++, it still avoids dynamic memory allocations and use of STL functions as the mBed devices sometimes have idiosyncratic memory models which make porting difficult. In any case, the library allows one to make memory use as predictable as possible.

21.1 Dependencies

- Eclipse Paho MQTT library - Provides a MQTT client library for mBed devices, check here for more information.
- EthernetInterface library - A mBed IP library over Ethernet.

21.2 How to use the library

Use the mBed Compiler to create your applications using this mBed C++ IBMIoTF Client Library. The mBed Compiler provides a lightweight online C/C++ IDE that is pre-configured to let you quickly write programs, compile and download them to run on your mbed Microcontroller. In fact, you don't have to install or set up anything to get running with mbed.

Refer to the step by step mBed C++ Client Library for IBM IoT Platform Recipe that shows how one can use this library to connect an ARM mBed NXP LPC 1768 microcontroller to the IoT Platform.

21.3 Constructor

The constructor builds the client instance, and accepts the following parameters:

- org - Your organization ID. (This is a required field. In case of quickstart flow, provide org as quickstart.)
- type - The type of your device. (This is a required field.)
- id - The ID of your device. (This is a required field.
- auth-method - Method of authentication (This is an optional field, needed only for registered flow and the only value currently supported is "token").
- auth-token - API key token (This is an optional field, needed only for registered flow).

These arguments create definitions which are used to interact with the IoT Platform service.

The following code block shows how to create a DeviceClient instance to interact with the IoT Platform quickstart service.

```
#include "DeviceClient.h"
....
....

// Set IoT Platform connection parameters
char organization[11] = "quickstart";     // For a registered connection, replace with your org
char deviceType[8] = "LPC1768";           // For a registered connection, replace with your device ty
char deviceId[3] = "01";                  // For a registered connection, replace with your device id

// Create DeviceClient
IoTF::DeviceClient client(organization, deviceType, deviceId);

// Get the DeviceID(MAC Address) if we are in quickstart mode and device id is not specified
if((strcmp(organization, QUICKSTART) == 0) && (strcmp("", deviceId) == 0))
{
      char tmpBuf[50];
      client.getDeviceId(tmpBuf, sizeof(tmpBuf));
}
....
```

As shown above, if the device id is not specified, the DeviceClient uses the MAC address of the device as device id and connects to the IoT Platform. The device code can use getDeviceId() method to retrieve the device id from the DeviceClient instance.

The following code block shows how to create a DeviceClient instance to interact with the IoT Platform Registered organization.

```
#include "DeviceClient.h"
....
....

// Set IoT Platform connection parameters
char organization[11] = "hrc178";
char deviceType[8] = "LPC1768";
char deviceId[3] = "LPC176801";
char method[6] = "token";
char token[9] = "password";

// Create DeviceClient
IoTF::DeviceClient client(organization, deviceType, deviceId, method, token);
....
```

21.4 Connecting to the IoT Platform

The device can connect to the IoT Platform by calling the connect function on the DeviceClient instance.

```
#include "DeviceClient.h"
....
....

// Create DeviceClient
```

```
IoTF::DeviceClient client(organization, deviceType, deviceId, method, token);

bool status = client.connect();
```

After the successful connection, the device can publish events to the IoT Platform and listen for commands.

Also, the device can query the status of the connection using the isConnected() method as follows,

```
#include "DeviceClient.h"
....
....

client.isConnected();
```

21.5 Publishing events

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

When an event is received by the IBM IoT Platform 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.

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.

21.5.1 Publish event using default quality of service

The below sample shows how to publish various data points of LPC1768 like x,y & z axis, joystick position, current temperature reading and etc.. to IoT Platform in JSON format.

```
boolean status = client.connect();

// Create buffer to hold the event
char buf[250];

// Construct an event message with desired datapoints in JSON format
sprintf(buf,
    "{\"d\":{\"myName\":\"IoT mbed\",\"accelX\":%0.4f,\"accelY\":%0.4f,\"accelZ\":%0.4f,
    \"temp\":%0.4f,\"joystick\":\"%s\",\"potentiometer1\":%0.4f,\"potentiometer2\":%0.4f}}",
    MMA.x(), MMA.y(), MMA.z(), sensor.temp(), joystickPos, ain1.read(), ain2.read());

status = client.publishEvent("blink", buf);
....
```

The complete sample can be found here.

21.5.2 Publish event using user-defined quality of service

Events can be published at higher MQTT quality of service levels, but these events may take slower than QoS level 0, because of the extra confirmation of receipt. Also quickstart flow allows only Qos of 0.

```
#include "MQTTClient.h"

boolean status = client.connect();

// Create buffer to hold the event
char buf[250];

// Construct an event message with desired datapoints in JSON format
sprintf(buf,
    "{\"d\":{\"myName\":\"IoT mbed\",\"accelX\":%0.4f,\"accelY\":%0.4f,\"accelZ\":%0.4f,
    \"temp\":%0.4f,\"joystick\":\"%s\",\"potentiometer1\":%0.4f,\"potentiometer2\":%0.4f}}",
    MMA.x(), MMA.y(), MMA.z(), sensor.temp(), joystickPos, ain1.read(), ain2.read());

status = client.publishEvent("blink", buf, MQTT::QOS2);
....
```

21.5.3 Handling the connection lost error during the event publish

When the publishEvent() method returns false, one can check the status of the connection and call reConnect() if the connection is lost,

```
#include "MQTTClient.h"

status = client.publishEvent("blink", buf, MQTT::QOS2);

if(status == false) {
    // Check if connection is lost and retry
    while(!client.isConnected())
    {
        client.reConnect();
        wait(5.0);
    }
}
....
```

The library does not store the events published during the unconnected state, and hence, the device needs to call the publishEvent() method again to send those events once the connection is reestablished.

21.6 Handling commands

When the device client connects, it automatically subscribes to any commands 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:

- command - name of the command invoked
- format - e.g json, xml
- payload

Following code defines a sample command callback function that processes the LED blink interval command from the application and adds the same to the DeviceClient instance.

```
#include "DeviceClient.h"
#include "Command.h"

// Process the command and set the LED blink interval
void processCommand(IoTF::Command &cmd)
{
    if (strcmp(cmd.getCommand(), "blink") == 0)
    {
        char *payload = cmd.getPayload();
        char* pos = strchr(payload, '}');
        if (pos != NULL) {
            *pos = '\0';
            char* ratepos = strstr(payload, "rate");
            if(ratepos == NULL)
                return;
            if ((pos = strchr(ratepos, ':')) != NULL)
            {
                int blink_rate = atoi(pos + 1);
                blink_interval = (blink_rate <= 0) ? 0 : (blink_rate > 50 ? 1 : 50/blink_rate);
            }
        }
    } else {
        WARN("Unsupported command: %s\n", cmd.getCommand());
    }
}

client.setCommandCallback(processCommand);

client.yield(10);  // allow the MQTT client to receive messages
....
```

The complete sample can be found here.

Note: The 'client.yield()' function must be called periodically to receive commands.

21.7 Disconnect Client

To disconnect the client and release the connections, run the following code snippet.

```
...
client.disconnect();
....
```

21.8 Samples

IBMIoTClientLibrarySample - A Sample code that showcases how to use IBMIoTF client library to connect the mbed LPC1768 or FRDM-K64F devices to the IBM Internet of Things Cloud service.

CHAPTER 22

MQTT Connectivity for Gateways

Important: This feature is currently available as part of a limited beta. Future updates may include changes incompatible with the current version of this feature. Try it out and let us know what you think

22.1 MQTT client connection

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

org_id.messaging.internetofthings.ibmcloud.com

22.1.1 Unencrypted client connection

Connect on port **1883**

Important: All information your gateway submits is being sent in plain text (including the authentication credentials for your gateway). We recommend the use of an encrypted connection whenever possible.

22.1.2 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 Platform 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)*.

22.2 MQTT client identifier

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

g:**orgId**:**typeId**:**deviceId**

- **g** idenfities your client as a gateway
- **orgId** is your unique organization ID, assigned when you sign up with the service. It will be a 6 character alphanumeric string.
- **typeId** is intended to be used as an identifier of the type of gateway connecting, it may be useful to think of this as analogous to a model number.
- **deviceId** must uniquely identify a gateway device across all gateways of a specific type, it may be useful to think of this as analogous to a serial number.

Note: You can use any scheme of your choice when assigning values for `typeId` and `deviceId`, however the following restrictions apply to both:

- Maximum length of 36 characters
- Must comprise only alpha-numeric characters (`a-z`, `A-Z`, `0-9`) and the following special characters:
  - dash (`-`)
  - underscore (_)
  - dot (`.`)

22.3 MQTT authentication

22.3.1 Username

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

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

22.3.2 Password

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

22.4 Publishing events

A gateway can publish events from itself and on behalf of any device connected via the gateway by using the following topic and substituting in the appropriate `typeId` and `deviceId` based on the intended origin of the event:

iot-2/type/**typeId**/id/**deviceId**/evt/**eventId**/fmt/**formatString**

22.4.1 Example

	typeId	deviceId
Gateway 1	mygateway	gateway1
Device 1	mydevice	device1

- Gateway 1 can publish it's own status events: `iot-2/type/mygateway/id/gateway1/evt/status/fmt/json`
- Gateway 1 can publish status events on behalf of Device 1: `iot-2/type/mydevice/id/device1/evt/status/fmt/js`

Important: The message payload is limited to a maximum of 4096 bytes. Messages larger than this will be rejected.

22.5 Subscribing to commands

A gateway can subscribe to commands directed at the gateway itself and to any device connected via the gateway by using the following topic and substituting in the appropriate `typeId` and `deviceId`:

iot-2/type/**typeId**/id/**deviceId**/cmd/**commandId**/fmt/**formatString**

The MQTT + wildcard can be used for `typeId`, `deviceId`, `commandId` and `formatString` to subscribe to multiple command sources.

22.5.1 Example

	typeId	deviceId
Gateway 1	mygateway	gateway1
Device 1	mydevice	device1

- Gateway 1 can subscribe to commands directed at the gateway: `iot-2/type/mygateway/id/gateway1/cmd/+/fmt/+`
- Gateway 1 can subscribe to commands sent to Device 1: `iot-2/type/mydevice/id/device1/cmd/+/fmt/+`
- Gateway 1 can subscribe any command sent to devices of type "mydevice": `iot-2/type/mydevice/id/+/cmd/+/fmt/+`

22.6 Managed Gateways

Support for device lifecycle management is optional, the device management protocol used by IoT Platform utilises the same MQTT connection that your gateway already uses for events and command control.

22.6.1 Quality of Service Levels and Clean Session

Managed gateways can publish messages with Quality of Service (QoS) level of 0 or 1. If QoS 1 is used, messages from the gateway will be queued if necessary. Messages from the gateway must not be retained messages.

The IoT Platform publishes requests with a QoS level of 1 to support queuing of messages. In order to queue messages sent while a managed gateway is not connected, the device should use `cleansession=false`.

Warning: If your managed gateway uses a durable subscription (cleansession=false) you need to be aware that device management commands sent to your gateway while it is offline will be reported as failed operations, however, when the gateway later connects those requests will be actioned by the gateway.
When handling failures it is important to take this into account if you are using durable subscriptions for your managed gateways.

22.6.2 Topics

A managed gateway is required to subscribe to two topics to handle requests and responses from IoT Platform:

- The managed gateway will subscribe to device management reponses on `iotdm-1/type/<typeId>/id/<deviceId>/response/+`
- The managed gateway will subscribe to device management requests on `iotdm-1/type/<typeId>/id/<deviceId>/+`

A managed gateway will publish to two topics:

- The managed gateway will publish device management responses on `iotdevice-1/type/<typeId>/id/<deviceId>/response/`
- The managed gateway will publish device management requests on `iotdevice-1/type/<typeId>/id/<deviceId>/`

The gateway is able to process device management protocol messages for both itself and on behalf other connected devices by using the relevant <typeId> and <deviceId>.

22.6.3 Message Format

All messages are sent in JSON format. There are two types of message.

1. **Request** Requests are formatted as follows:

```
{  "d": {...}, "reqId": "b53eb43e-401c-453c-b8f5-94b73290c056" }
```

   - `d` carries any data relevant to the request
   - `reqId` is an identifier of the request, and must be copied into a response. If a response is not required, the field should be omitted.

2. **Response** Responses are formatted as follows:

```
{
    "rc": 0,
    "message": "success",
    "d": {...},
    "reqId": "b53eb43e-401c-453c-b8f5-94b73290c056"
}
```

   - "rc" is a result code of the original request.
   - `message` is an optional element with a text description of the response code.
   - `d` is an optional data element accompanying the response.
   - `reqId` is the request ID of the original request. This is used to correlate responses with requests, and the device needs to ensure that all request IDs are unique. When responding to IoT Platform requests, the correct `reqId` value must be sent in the response.

CHAPTER 23

MQTT

The primary mechanism that devices and applications use to communicate with the IBM Watson IoT platform 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.

23.1 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
- MQTT Connectivity for Gateways

23.2 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.

23.2.1 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.

23.2.2 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.

23.2.3 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.

23.3 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 false, a subscriber will start receiving messages from the buffer. If clean session is set to true, 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.

23.4 Message Payload

The Watson IoT platform supports sending and recieving messages in any format.

23.4.1 Watson IoT Platform Maximum Message Payload Size

The maximum payload size is 4 kilobytes (kB). If messages with payloads larger than this limit are sent, the client will be disconnected and the following log message will appear in the Diagnostic Logs if the client is a device:

Closed connection from x.x.x.x. The message size is too large for this endpoint.

CHAPTER 24

Device Model

The device model describes the metadata and management characteristics of a device. The device database in the Watson IoT platform is the master source of device information. Applications and managed devices are able to send updates to the database such as a location or the progress of a firmware update. Once these updates are received by the Watson IoT platform, the device database is updated, making the information available to applications.

Note: With the exception of the management extension, the entire device model is available for both managed and unmanaged devices. However, an unmanaged device can not directly update its device model in the database.

24.1 Device Identification

Every device has a typeId and deviceId attribute. Typically, the typeId represents the model of your device, whilst the deviceId can represent its serial number. Within your Watson IoT platform organization, the combination of typeId and deviceId must be unique for each device.

In addition, the Watson IoT platform constructs a further identifier for each device based on your organizationId, the device's typeId and the device's deviceId. This identifier - the clientId - allows you to identify unequivocally an individual device.

The characters which can be used in these identifiers are restricted so they can be used straightforwardly in communication protocols and REST APIs. There are a number of optional device identifiers for use with the device management protocol. These identifiers are:

- deviceInfo.manufacturer
- deviceInfo.serialNumber
- deviceInfo.model
- deviceInfo.deviceClass
- deviceInfo.description

For more information on the identifiers and descriptions of their comparative identifiers in other device management standards, see Device Model Attributes Reference.

24.2 Identifiers and Device Type

Each device connected to the Watson IoT platform is associated with a device type. Device types are intended to be groups of devices which share characteristics or behaviour.

A device type has a set of attributes. When a device is added to the Watson IoT platform, the attributes in its device type are used as a template overridden by device-specific attributes. For example, the device type could have a value for the `deviceInfo.fwVersion` attribute reflecting the firmware version at the time of manufacture, and this value would be copied from the device type into the devices as they are added. However, if a device was added which already had a value for `deviceInfo.fwVersion`, it would not be overridden by the device type.

When a device type is updated, future registered devices will reflect the modified device type template. However, existing devices of this device type are unaffected and remain unchanged.

24.3 Attributes

The table below shows the list of attributes which can apply to devices in the Watson IoT platform. Additionally, italicized attributes can also apply to device types.

- API/DMA: Can be updated by API/Device Management Agent
  - R: Read only
  - W: Write and read
  - A: Append

Attribute	Type	Description	API	DMA
clientId	string	The client ID used with MQTT connections	R	
typeId	string	Device type	R	
deviceId	string	Device ID	R	
classId	string	Class of device (“Device” or “Gateway”)	R	
gatewayTypeId	string	Type ID of the gateway this device is using	R	
gatewayId	string	Device ID of the gateway this device is using	R	
status.alert	boolean	Whether the device has an alert	W	
deviceInfo.serialNumber	string	The serial number of the device	W	W
deviceInfo.manufacturer	string	The manufacturer of the device	W	W
deviceInfo.model	string	The model of the device	W	W
deviceInfo.deviceClass	string	The class of the device	W	W
deviceInfo.description	string	The descriptive name of the device	W	W
deviceInfo.fwVersion	string	The firmware version currently known to be on the device	W	W
deviceInfo.hwVersion	string	The hardware version of the device	W	W
deviceInfo.descriptiveLocation	string	The descriptive location, such as a room or building number, or a geopgraphical region	W	W
metadata	complex	Free-form metadata	W	W
added.auth.id	string	ID that added the device	R	
added.dateTime	string	ISO8601 date-time: Date and time the device was added	R	
refs.diag.errorCodes	string	URI of diag extension for error codes, if present	R	
refs.diag.logs	string	URI of diag extension for logs, if present	R	
refs.location	string	URI of location extension, if present	R	
refs.mgmt	string	URI of mgmt extension, is present	R	

24.4 Extended Attributes

In addition to core attributes listed above, there are additional attributes which are treated as extensions to the core device model. Simple queries about the device return information from the core device model, but not the extensions. Information from the extensions must be specifically requested.

Extension Name	Prefix of attributes	Purpose
Diagnostics	diag	Error logs and diagnostic information
Location	location	Location of the device, potentially updated regularly
Device management	mgmt	Device management actions, e.g. firmware update

24.4.1 Diagnostics Extension

The diagnostics attributes are optional, and only present for devices with error log information. These attributes are intended for diagnosing device problems, not troubleshooting connectivity to the Watson IoT platform. In order to retrieve the information in these attributes, it must be queried separately, because the information stored in these attributes could potentially be very large.

Diagnostic log information is an array of entries which can have entries appended using an API, however, this can cause earlier entries to be lost, to keep the size of diagnostic logs manageable. Each entry consists of a message, an indication of severity, a timestamp and an optional byte-array of data.

Attribute	Type	Description	API	DMA
diag.errorCodes[]	array of integer(s)	Array of error codes	A	A
diag.log[]	array	Array of diagnostic data	A	A
diag.log[].message	string	Diagnostic message		
diag.log[].timestamp	string	ISO8601 date-time: Date and time of log entry		
diag.log[].logData	string	byte: Diagnostic data, base-64 encoded		
diag.log[].severity	number	Severity of message, 0: informational, 1: warning, 2: error		

24.4.2 Location Extension

These attributes are optional and only present for devices with location information. The location information is stored separately in order to allow the use of storage mechanisms better suited to dynamic information in the event of frequently updated information, for example, in the case of a mobile device.

For solutions which place significant importance on frequent location updates, it is expected that the location would be treated as part of the device's event payload, enabling higher update rates, simple historical storage, and analytics.

Attribute	Type	Description	API	DMA
location.longitude	number	Longitude in decimal degrees using WGS84	W	W
location.latitude	number	Latitude in decimal degrees using WGS84	W	W
location.elevation	number	Elevation in metres using WGS84	W	W
location.measuredDateTime	string	ISO8601 date-time: Date and time of location measurement	W	W
location.updatedDateTime	string	ISO8601 date-time: Date and time	R	
location.accuracy	number	Accuracy of the position in metres	W	W

24.4.3 Device Management Extension

The `mgmt.` attributes are only present for managed devices. When a managed device becomes dormant, it becomes unmanaged and the `mgmt.` attributes are deleted. The `mgmt.` attributes are set by the Watson IoT platform as a result of processing device management requests. These attributes cannot be directly written using the API.

Devices have a management lifecycle, defined by their status as managed devices. The device management agent on the device is responsible for sending a Manage Device request using the device management protocol. To deal with defunct devices in large device populations, a managed device can be set to send a Manage Device request regularly, allowing the Watson IoT platform to notice when a device has become dormant. To facilitate this functionality, the Manage Device request has an optional lifetime parameter, When the Watson IoT platform receives a Manage Device request with a lifetime, it calculates the time before which another Manage Device request is required and stores it in the "mgmt.dormantDateTime" attribute.

Attribute	Type	Description	API	DMA
mgmt.dormant	boolean	Whether the device has become dormant	R	
mgmt.dormantDateTime	string	ISO8601 date-time: Date and time at which the managed device will become dormant	R	
mgmt.lastActivityDateTime	string	ISO8601 date-time: Date and time of last activity, updated periodically	R	
mgmt.supports.deviceActions	boolean	Whether the device supports Reboot and Factory Reset actions	R	
mgmt.supports.firmwareActions	boolean	Whether the device supports Firmware Download and Firmware Update actions	R	
mgmt.firmware.version	string	The version of the firmware on the device	R	W
mgmt.firmware.name	string	The name of the firmware to be used on the device	R	W
mgmt.firmware.uri	string	The URI from which the firmware image can be downloaded	R	W
mgmt.firmware.verifier	string	The verifier such as a checksum for the firmware image to validate its integrity	R	W
mgmt.firmware.state	number	Indicates the state of firmware download	R	W
mgmt.firmware.updateStatus	number	Indicates the status of the update	R	W
mgmt.firmware.updatedDateTime	string	ISO8601 date-time: Date of last update	R	

CHAPTER 25

External Service Integrations

25.1 Overview

External service integration allows you to bind metadata or function supported by another service to your Watson IoT account.

25.2 Jasper

Jasper is an administration and management platform for SIM devices. Jasper has been integrated into the IBM Watson IoT platform dashboard, making it possible to administer Jasper devices through your Watson IoT platform organization dashboard.

Important: Jasper integration is currently available as part of a limited beta. Future updates may include changes incompatible with the current version of this feature.

25.2.1 Supported Operations

The built-in Jasper integration provided by our platform provides support for the following Jasper operations:

- View overall Jasper data
  - Shows: Status, Rate Plan, month-to-date data usage, month-to-date SMS usage, month-to-date voice usage, overage limits, date added, and date modified.
- Change SIM activation state.
  - Select from: Inventory, Activation Ready, Activated, Deactivated, and Retired.
- View SIM Usage
  - Shows: Cycle start date, billable and total data, billable and total SMS, billable and total voice.
  - The cycle start date can be set using a YYYY-MM-DD format.
- Send SMS to SIM
- Change rate plan

These operations can be accessed in the device drilldown of a Jasper connected device after the configuration steps below are completed.

25.2.2 Configuration

In order to connect your Watson IoT Platform platform organization with your Jasper account, there are two stages of configuration which must first be performed. The first stage of configuration is Organization Configuration.

Extension Enablement

To enable Jasper integrtion with your Watson IoT organization follow these steps:

1. When in your Watson IoT platform dashboard, click the wrench icon on the right to open the Configuration Settings.
2. Scroll down the to the Extensions section, and set Jasper to 'On'.
3. Enter your Jasper username, password, license key, and API endpoint.

Device Configuration

Devices which are connected to both your Watson IoT platform organization and your Jasper account can be configured to display data from Jasper in the Watson IoT platform dashboard. Jasper configuration cannot be applied as part of the Add Device process, only already connected devices can be configured with Jasper. To configure your Jasper-connected devices, follow these steps:

1. In the devices tab of your Watson IoT platform dashboard, find the Jasper-connected device to be configured.
2. Select the device to open the Device Drilldown view.
3. Scroll down to the 'Extension Configuration section. The extension configuration must be entered as JSON in the following format:

```
{
    "jasper": {
        "iccid": "string"
    }
}
```

4. When this has been entered, click 'Confirm changes' to save the configuration.

If the Organization configuration has been completed correctly, the Extensions section should now appear below the Extensions Configuration section in the Device Drilldown.

CHAPTER 26

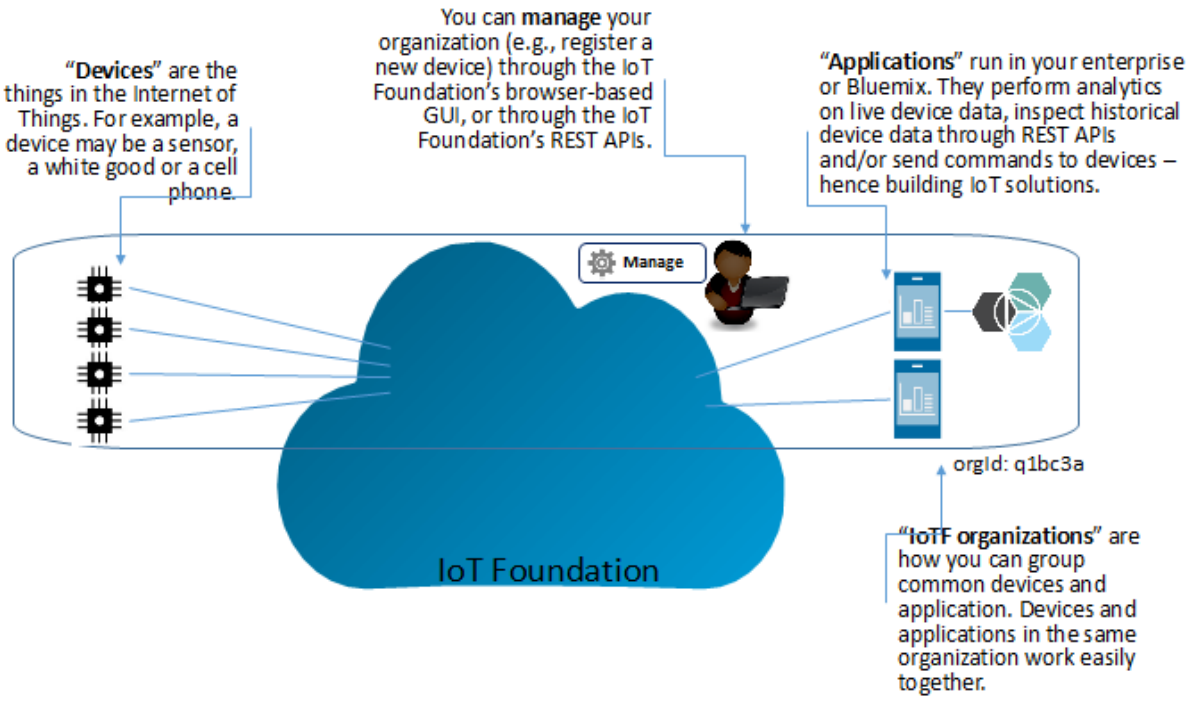
Securing the Watson IoT platform

The IBM Watson IoT platform 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.

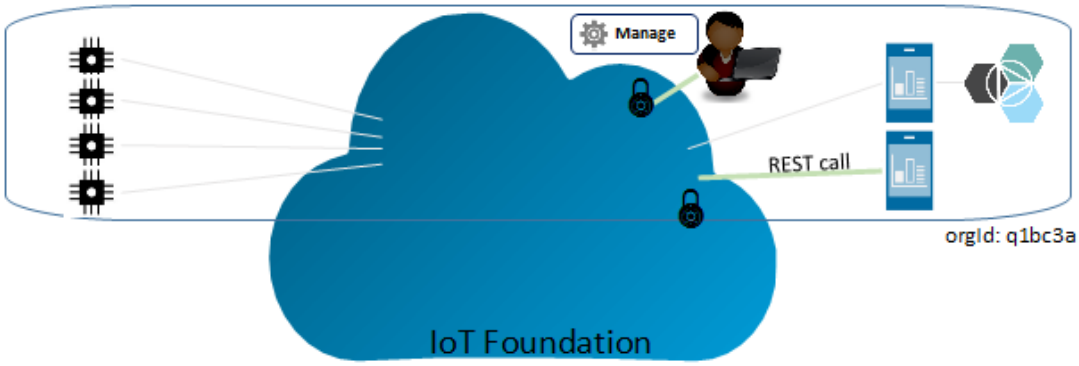
26.1 Terminology



26.2 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 Watson IoT platform.

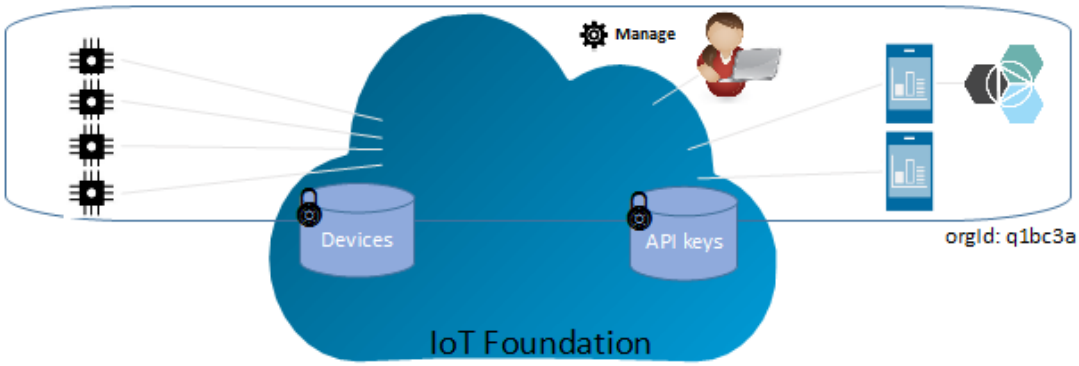
- GUI: authenticated via your IBM ID.
- REST API: once you create an API key through the GUI, you can use this to make authenticated REST calls against your organization.



26.3 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 Watson IoT platform is compromised.

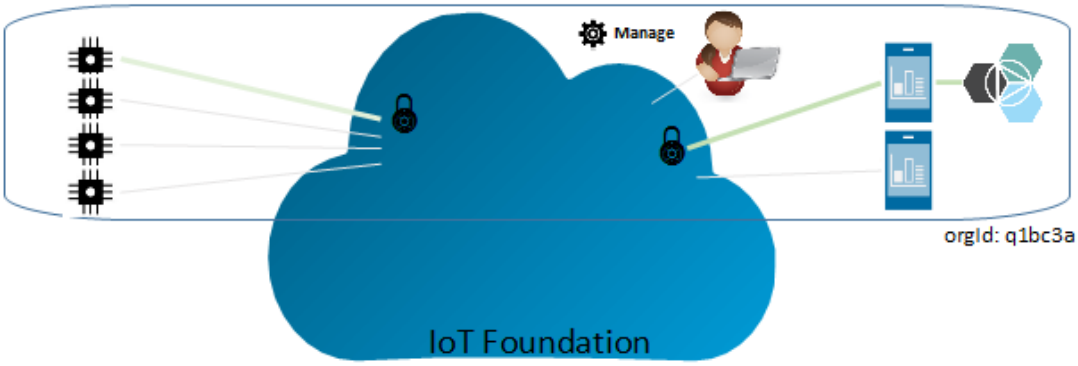
Device credentials and API keys can be individually revoked if they are compromised.



26.4 How do we ensure your devices connect securely to the Watson IoT platform?

- 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.



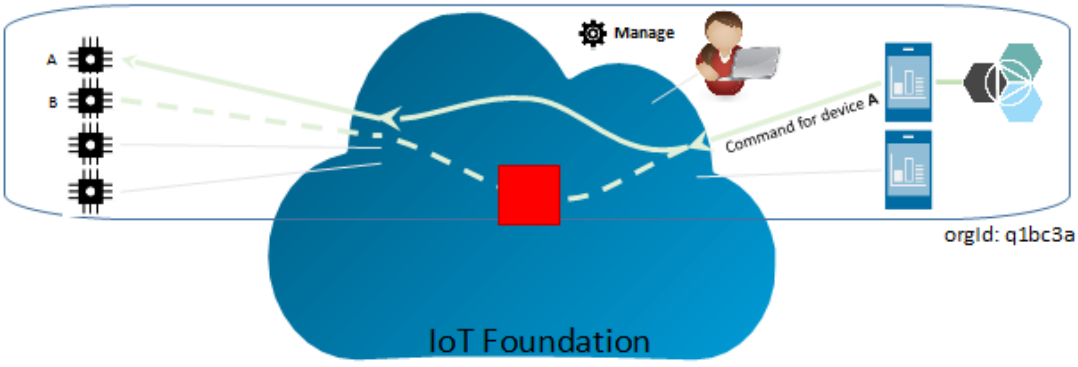
26.5 How do we prevent data leaking between devices?

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 Watson IoT platform. This prevents devices from being able to impersonate another device.

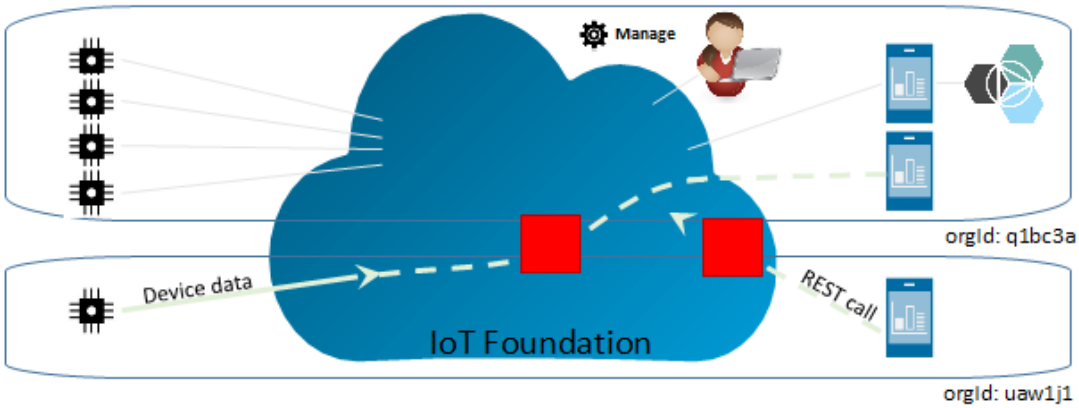
The only way to impersonate another device is by obtaining compromised security credentials for the device.



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

26.6 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 Watson IoT platform 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 27

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.