EdgeX Foundry: A hands-on tutorial

A practical guide to getting started with open source IoT

Author:	Jonas Werner
Date:	2020-08-26
Version:	1.0
EdgeX Foundry version:	Geneva

Table of contents

Disclaimer

I'm a happy enthusiast and don't claim to be an expert on EdgeX Foundry or any other topic covered in this document. However, I have over 22 years of professional IT experience and the last two years I've worked with EdgeX Foundry. On the way I've picked up a few things. This document attempts to be a practical guide for newcomers to get started.

This guide is provided as-is without any warranties or support. Use at your own risk.

Scope and format

Since this is meant to be a beginners guide to EdgeX Foundry it assumes nothing but some basic Linux command line experience. While the level of detail is aimed to support new Linux users, the content will be applicable also to experienced users who wish to get into open source IoT and perhaps who want to learn more about technologies such as docker-compose, etc.

Some of the topics covered:
- Installation of EdgeX Foundry
- Starting / stopping microservices
- How to add / enable services
- Interacting with EdgeX using Postman, cURL and Python
- Creating devices (sources of sensor data)
- Sending data to EdgeX using REST
- Exporting a stream of data using MQTT
- How to issue commands from EdgeX to devices
- Creating rules
- Debug flags and container logs

Learning about EdgeX Foundry can also be an excellent way to learn about:
- Docker and containers
- Microservices
- Launching multiple containers in a group with docker-compose
- How to interact with REST API's
- Postman, Curl, virtual Python environments and other tools of the IT trade

Note:
- When the instructions refers to the IP address of the Linux VM which EdgeX Foundry is installed on it will be referred to as: <edgex ip>. Substitute this with the actual IP address for any commands, without the "<>".
- Commands to be entered into the terminal are written and highlighted in `this format`
- All code and configuration files referenced will be shown with direct download links. However, all can be cloned in one go with git from if that is preferred: https://github.com/jonas-werner/EdgeX_Tutorial.git

EdgeX Foundry introduction

EdgeX Foundry is an open source IoT solution ideally used for ingesting data from multiple sources and then forwarding that data to a central system. It natively speaks multiple protocols used by IoT devices, like BACNET, OPC-UA, MQTT and REST. It can also be configured to match individual data formats used by devices from different vendors by using device profiles. EdgeX Foundry is made up of a collection of micro services, each of which runs in a container. Microservices communicate with each other over REST API interfaces.
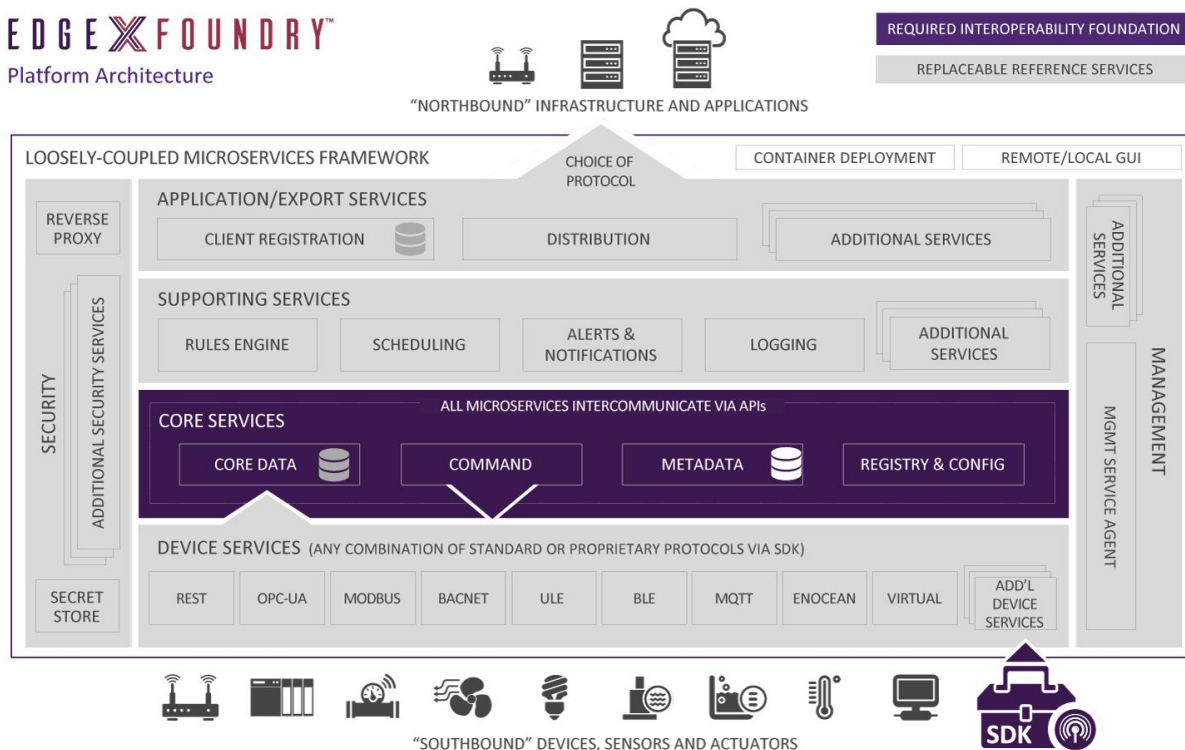
EdgeX Foundry can convert source data from proprietary data formats into XML or JSON, encrypt, compress and finally forward that data to an external source over MQTT or other protocol. Data is normally not retained long-term by EdgeX Foundry itself.

Depending on protocol, sending commands is also supported. Therefore it's possible to use EdgeX as an intermediary when wanting to communicate with a device over for example BACNET but not wanting to build support for that protocol oneself. By using the REST API provided by the command service, commands can be automatically translated by EdgeX from REST into the correct protocol and format expected by the end device.

Rules can be used to create logic for triggering actions based on input. For example, if value A goes above X, execute a pre-set command.

Installation of EdgeX Foundry would generally be done close to the sensors / data being generated. For example on an edge gateway appliance. There could be thousands of these, each with its own EdgeX installation, ingesting, converting and forwarding data to a central location. While EdgeX runs as a collection of containerized microservices it doesn't currently support k8s. Note that since the overhead for k8s can be prohibitive for low-powered edge nodes, lack of k8s support isn't necessarily a concern.

Installation

Installation is straight forward but there are a few items to be aware of before getting started with the tutorial. Note that the prerequisites are for the tutorial and should not be interpreted as limitations of EdgeX. EdgeX could run on essentially any platform supporting Docker and docker-compose.

Tutorial prerequisites

- Ubuntu 20.04 (preferably a VM)
    - Any Linux OS supporting docker and docker-compose should work but the tutorial uses Ubuntu 20.04 and commands will reflect this
- Internet access
    - For downloading container images and sending data via MQTT
- Familiarity with Linux, general terminal commands and text editing tools
- Optional:
    - Raspberry Pi (for those who want to use a DHT sensor to send data to EdgeX)
    - An IDE like VS Code, Atom or similar to edit code and settings files

Note:
If the Linux OS hosting EdgeX is running as a VM and if there is a plan to use a Raspberry Pi for sensors: Bridge the VM network interface to give it an IP on the local network. This will make it possible for the Pi to communicate with the EdgeX installation directly (without requiring port forwarding).
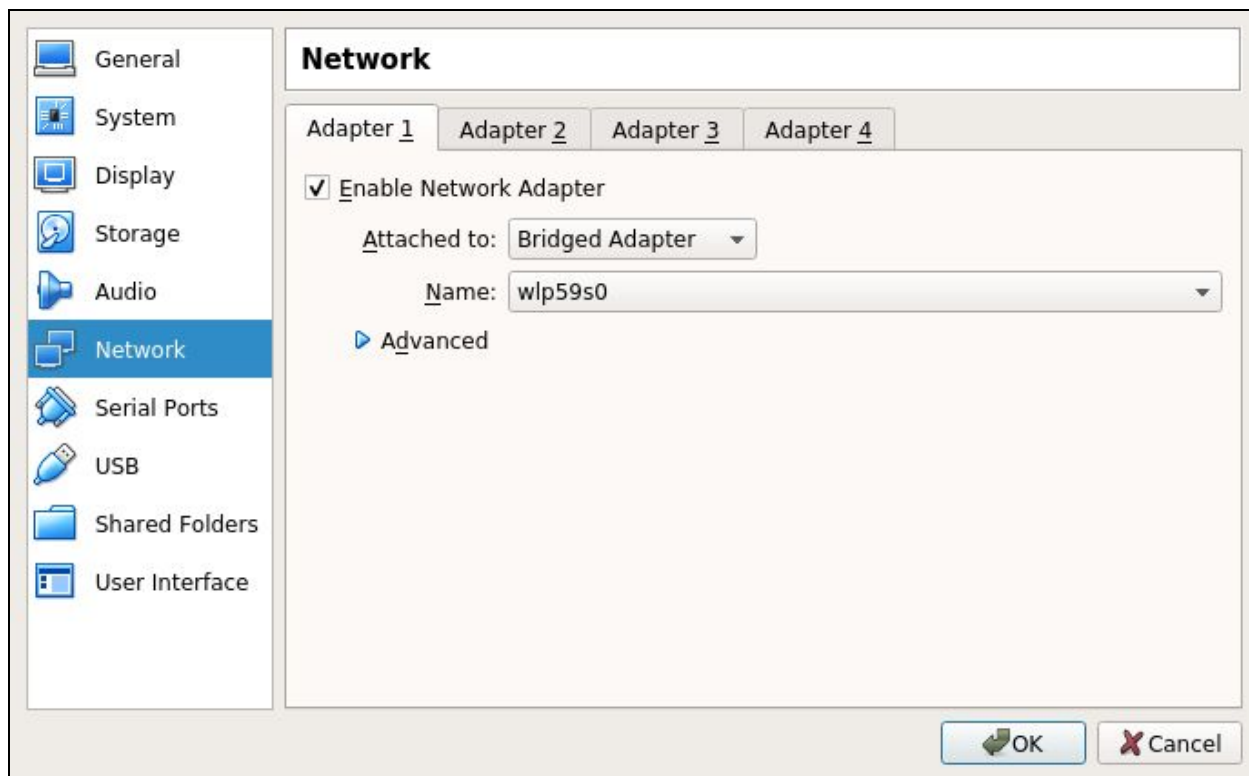
Example of bridged network adapter in Virtualbox

Installing Docker and docker-compose

1. SSH to the VM where EdgeX Foundry will be installed
2. Update system

```
sudo apt update
sudo apt upgrade -y
```

3. Install Docker-CE

```
sudo apt install apt-transport-https ca-certificates curl
software-properties-common -y
```

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
apt-key add -
```

Note: Match "**focal**" below with your distribution if different from 20.04 (check with "lsb_release -a" if unsure about the version):

```
sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu focal stable"
```

```
sudo apt update
sudo apt install docker-ce -y
sudo usermod -aG docker ${USER}
```

4. Log out and back in again so the new permissions applied by "usermod" can take effect
5. Install docker-compose
```
sudo apt install docker-compose -y
```

Installing EdgeX Foundry

The microservices making up EdgeX Foundry are controlled by a docker-compose file in YAML format. It specifies how each microservice should run, its ports, volumes and dependencies.

Docker-compose manages the containers in the YAML file as a group. Commands starting with "docker-compose" are context sensitive and need to be executed in the same folder as the docker-compose.yml file.

1. Create a directory for the EdgeX Foundry docker-compose.yml file (Geneva release):
```
mkdir geneva
cd geneva
```
2. Use wget to download the docker-compose.yml file

```
wget
https://raw.githubusercontent.com/jonas-werner/EdgeX_Tutorial/m
aster/docker-compose_files/docker-compose_step1.yml
```

```
cp docker-compose_step1.yml docker-compose.yml
```

Note: The official docker-compose files are located on Github as per the link below but will not be used for this tutorial as we want to minimize editing of yaml for beginners: Official releases: https://github.com/edgexfoundry/developer-scripts/tree/master/releases

3. Pull the containers and list the newly downloaded images
```
docker-compose pull
docker image ls
```

```
vagrant@edgex-geneva-03:~/EdgeX_Tutorial/docker-compose_files$ docker image ls
REPOSITORY                                    TAG              IMAGE ID
edgexfoundry/docker-device-rest-go            1.1.1            5d8734bd1a55
edgexfoundry/docker-support-scheduler-go      1.2.1            8ff56ef98c94
edgexfoundry/docker-sys-mgmt-agent-go         1.2.1            8069d55f1860
edgexfoundry/docker-support-notifications-go  1.2.1            81bfc8eb1799
edgexfoundry/docker-core-command-go           1.2.1            c9e88b3bbd8b
edgexfoundry/docker-core-metadata-go          1.2.1            a33bfd480526
edgexfoundry/docker-core-data-go              1.2.1            f50adbca9cb2
edgexfoundry/docker-app-service-configurable  1.2.0            5b8a3e288cc1
edgexfoundry/docker-edgex-consul              1.2.0            9af65397ea6d
redis                                         5.0.8-alpine     5c5637d8a823
vagrant@edgex-geneva-03:~/EdgeX_Tutorial/docker-compose_files$
```

Starting EdgeX Foundry

1. Start EdgeX Foundry using docker-compose
    Make sure the commands below are executed in the same folder as the YAML file. Don't
    forget the "-d" at the end or the terminal will be flooded by log output from ALL the
    microservices. If this happens, press "CTRL+C" to shut down EdgeX Foundry.
    ```
    docker-compose up -d
    ```

2. View the running containers
    ```
    docker-compose ps
    ```

    Note that the ports used by EdgeX are listed for each container. These ports are defined
    in the docker-compose.yml file along with many other settings.

```
vagrant@edgex-geneva-03:~/EdgeX_Tutorial/docker-compose_files$ docker-compose ps
            Name                         Command              State
-----------------------------------------------------------------------------------------
edgex-app-service-configurable-rules  /app-service-configurable  ...  Up   48095/tcp, 0.0.0.0:48100->
edgex-core-command                    /core-command -cp=consul.h ...  Up   0.0.0.0:48082->48082/tcp
edgex-core-consul                     edgex-consul-entrypoint.sh ...  Up   8300/tcp, 8301/tcp, 8301/u
                                                                           0.0.0.0:8500->8500/tcp, 86
edgex-core-data                       /core-data -cp=consul.http ...  Up   0.0.0.0:48080->48080/tcp,
edgex-core-metadata                   /core-metadata -cp=consul. ...  Up   0.0.0.0:48081->48081/tcp
edgex-device-rest                     /device-rest-go --cp=consu ...  Up   0.0.0.0:49986->49986/tcp
edgex-redis                           docker-entrypoint.sh redis ...  Up   0.0.0.0:6379->6379/tcp
edgex-support-notifications           /support-notifications -cp ...  Up   0.0.0.0:48060->48060/tcp
edgex-support-scheduler               /support-scheduler -cp=con ...  Up   0.0.0.0:48085->48085/tcp
edgex-sys-mgmt-agent                  /sys-mgmt-agent -cp=consul ...  Up   0.0.0.0:48090->48090/tcp
vagrant@edgex-geneva-03:~/EdgeX_Tutorial/docker-compose_files$
```

3. Also compare the previous output with the output from the docker ps command
    ```
    docker ps
    ```

Basic interaction

Consul

View the status of the microservices via the Consul web interface. Use a browser to access:
http://<edgex ip>:8500/ui/dc1/services

Consul can also be used to change configuration settings, for example switching on debugging.

cURL

1. Interact with EdgeX using curl. In this case we list the devices registered:
   ```
   curl http://<edgex ip>:48082/api/v1/device
   ```
   It may take a few seconds to complete and will result in some rather difficult to read output. Let's improve that.

2. Install jq to do pretty formatting of JSON output
   ```
   sudo apt install jq
   ```

3. Issue the same curl command as before but add a pipe and the jq command:
   ```
   curl http://<edgex ip>:48082/api/v1/device | jq
   ```
   This will result in much prettier formatting, making it easy to see the sample devices which have already been created

```
{
    "id": "f14e3e10-de12-42ec-bff6-c32e48e073f0",
    "name": "sample-image",
    "adminState": "UNLOCKED",
    "operatingState": "ENABLED",
    "labels": [
        "rest",
        "binary",
        "image"
    ]
}
```

Tip: Note the port used in the curl command. Going forward it is helpful to use "docker-compose ps" to find out which service we are interacting with by looking for the port number in the output.

Postman

Postman is an excellent tool for interacting with REST APIs in a graphical manner. It's also great since it saves the history for later reference and it also comes with the ability to save frequently used queries into collections. Install the stand-alone version of Postman rather than the Google Chrome plug-in, since the plug-in is outdated.

1. After installation, disable certificate verification:
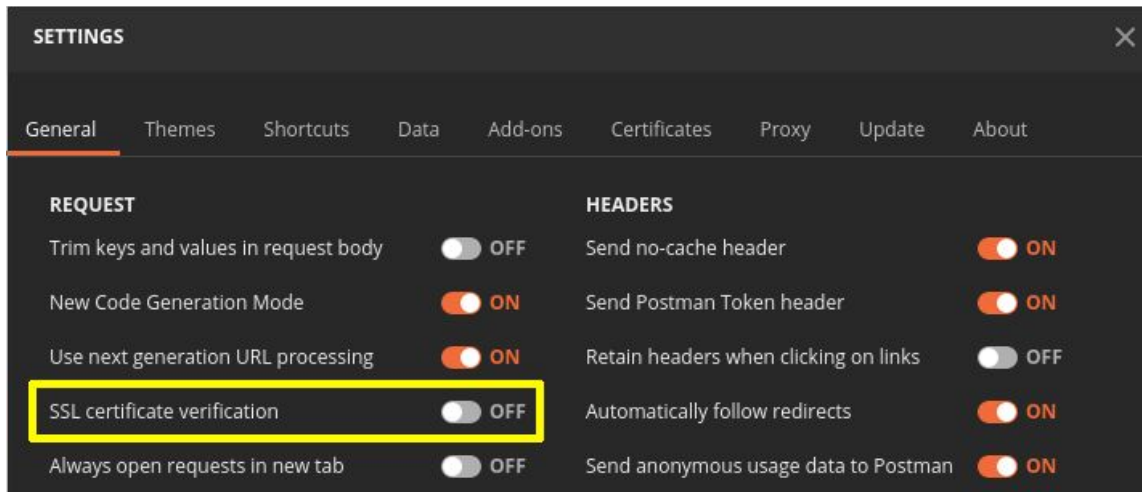
It's also possible to change the theme under the "Themes" tab

2. Set the method to "GET"
3. Enter the URI to be used:
   `http://<edgex ip>:48082/api/v1/device`
4. Push "Send"
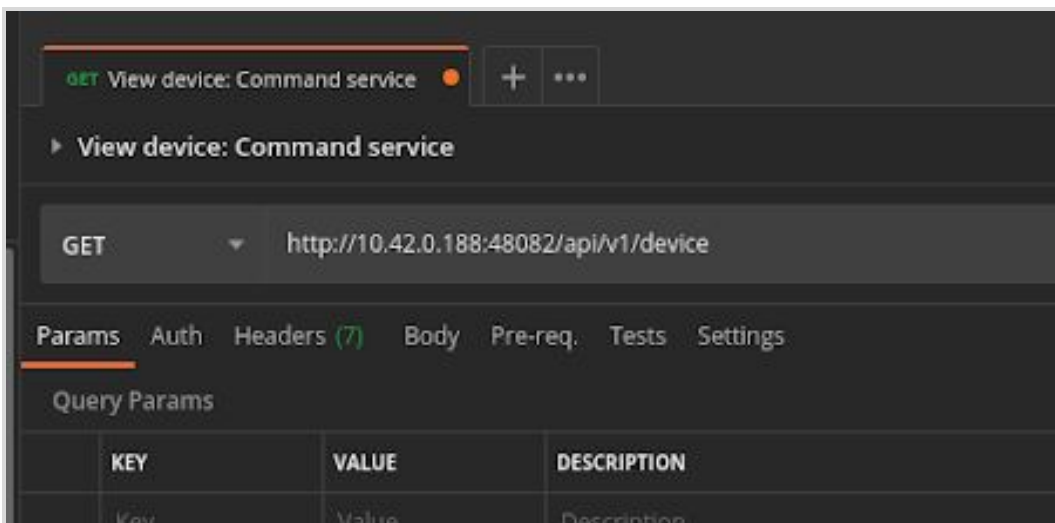
5. The devices registered with EdgeX Foundry will be listed in the results pane. This is the exact same output as was received back when using cURL

Stopping EdgeX Foundry

The commands in this section need to be issued from the folder containing the docker-compose.yml file ("geneva" in this example).

1. To just stop the containers:
   ```
   docker-compose stop
   ```

2. To stop and remove the containers:
   ```
   docker-compose down
   ```

3. To stop and remove containers + volumes (the original images will remain):
   ```
   docker-compose down -v
   ```

Editing the docker-compose.yml file

This section describes how to edit the docker-compose.yml file on the VM running EdgeX Foundry. This is in preparation for adding additional UI elements in the next section: Adding additional graphical user interfaces. No editing will actually be done in this section.

Controlling micro services

The docker-compose.yml file contains the information for each of the micro services that makes up EdgeX and states how they should be run, including network, ports and volumes. Many sections are present but will be commented out. This is especially true for device services supporting their respective protocols. Remove the commenting to enable these services to start together with the rest of EdgeX. It is also possible to add new entries to expand the capabilities of an EdgeX Foundry installation.

If downloading an official docker-compose.yml file, note that the IP addresses for each service is set to the loopback address ("127.0.0.1"). This will make containers bind to the loopback address only, and be accessible only from the host on which EdgeX is running. To make EdgeX accessible from the outside, remove or change the IP from "127.0.0.1" to "0.0.0.0". This will make the containers listen on all IP addresses available. The docker-compose.yml files used in this tutorial have already been updated to allow for outside access.

Edit from the command line

In the "geneva" directory, use a console text editor like vi, vim or nano to modify the file. For example:

```
vi docker-compose.yml
```

Nano is frequently considered easier for new users. For those who are brave the vi editor is extremely powerful and useful. vi editor cheat sheet for reference:
http://www.atmos.albany.edu/daes/atmclasses/atm350/vi_cheat_sheet.pdf

Edit graphically

It can be easier to edit using an IDE like VS Code or Atom. To edit the file directly, use Filezilla or a similar tool supporting SFTP (secure FTP) to edit these files over an SSH session.

1. Connect to the VM over SFTP (use the SSH username / password and port 22)
2. Enter the "geneva" folder
3. Right-click the "docker-compose.yml" file and choose "View/Edit"
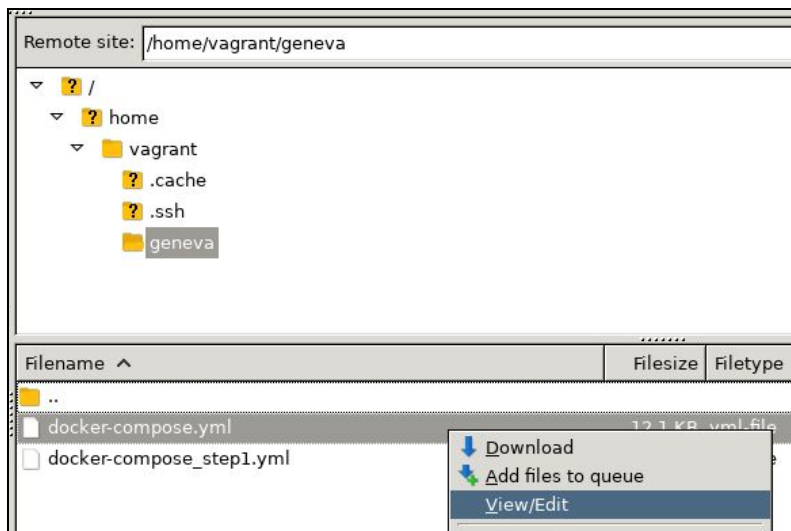
4. When asked, select to open with the tool desired, for example VS Code or Atom
5. Perform the edits required (will be done in the next section to add new functionality)
6. Save the file
7. Switch back to Filezilla again. It will now ask if you wish to finish editing and upload the file. Select "yes". Also select to delete the local copy.
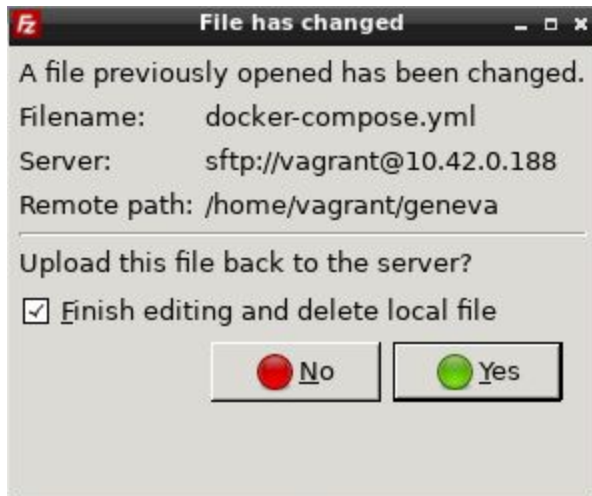
Optional: Adding additional graphical user interfaces

There are multiple UI's that can be added to EdgeX Foundry. To do this, simply add entries for them in the docker-compose.yml file. In this case Portainer will be added as a way to view and interact with the containers in a graphical manner. In addition, the EdgeX UI written in Golang will also be added as a way to view device services and other information via a web browser rather than the command line.

Portainer

1. Open the docker-compose.yml file. Under the volumes section at the beginning of the file, add an entry for portainer as per the below:

```
volumes:
  db-data:
  log-data:
  consul-config:
  consul-data:
  portainer_data:
```

2. Under the "services" section, add the following entry for Portainer:

```
portainer:
  image:  portainer/portainer
  ports:
    - "0.0.0.0:9000:9000"
  container_name: portainer
  command: -H unix:///var/run/docker.sock
  volumes:
```
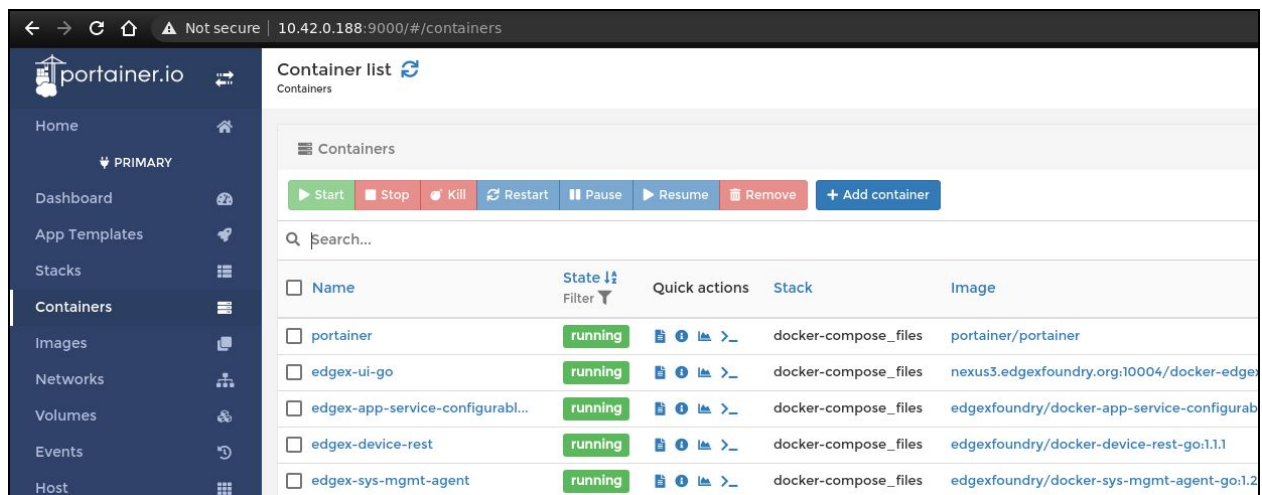
```
        - /var/run/docker.sock:/var/run/docker.sock:z
        - portainer_data:/data
```

Note: Be careful to match the indentation with the other services entries. If in doubt, compare to entries above or below to make sure the indentation matches.

3. Save and exit the editor
4. Switch to Filezilla and click "yes" to save and upload
5. Start (or restart) EdgeX Foundry
6. The new container image will be downloaded and started

```
vagrant@edgex-tutorial:~/geneva$ docker-compose up -d
Creating network "geneva_default" with the default driver
Creating volume "geneva_portainer_data" with default driver
Pulling portainer (portainer/portainer:)...
latest: Pulling from portainer/portainer
d1e017099d17: Pull complete
717377b83d5c: Downloading [========================================>    ]  21.16MB/23.92MB
```

7. Portainer can now be accessed in a browser at:
   http://<edgex ip>:9000

Portainer interface for a graphical view of the containers and images that makes up EdgeX

EdgeX Golang UI

1. Open the docker-compose.yml file in an editor
2. Add an entry for the golang ui under the "services" section as per the below:

```
ui:
    container_name: edgex-ui-go
    hostname: edgex-ui-go
    image:
nexus3.edgexfoundry.org:10004/docker-edgex-ui-go:master
```

```
networks:
  edgex-network: null
ports:
- "0.0.0.0:4000:4000/tcp"
read_only: true
```

Note: Be careful to match the indentation with the other services entries. If in doubt, compare to entries above or below to make sure the indentation matches.

3. Save and exit the editor
4. Start EdgeX Foundry
   Tip: It's possible to execute "docker-compose up -d" again even it it is already running
5. The EdgeX UI can now be accessed in a browser at:
   http://<edgex ip>:4000

   If asked, use "admin" / "admin" for username / password.

The optional Go UI for EdgeX

Creating a device

A device could be any type of edge appliance which is generating or forwarding data. It could be an edge gateway in a factory with sensors of its own or an industrial PC hooked up to a PLC or any other device.

Creating, or registering, a device in EdgeX Foundry is required for EdgeX to:
- Become aware of the existence of the device
- Be able to receive data from the device
- Be able to send commands to the device (if it supports commands)
- Understand what type of data the device will generate
- Understand how and in what format the device sends data / receives commands. For example:
  - What protocol is used?
  - What types of data is supported (temp, humidity, vibrations/sec, etc.)?
  - What format does the data come in (Int64, Str, etc.)?

In this tutorial two types of devices will be created:
1. Sensor cluster generating temperature and humidity data
2. Generic device with a REST interface, supporting commands

Two methods of device creation will be covered:
- Manual: The Sensor cluster will be created by issuing individual REST commands
- Scripted: The Generic device will be created instantly using a Python script

Introduction to Device Profiles

EdgeX incorporates device profiles as a way of easily adding new devices. A device profile is essentially a template which describes the device, its data formats and supported commands. It is a text file written in YAML format which is uploaded to EdgeX and later referenced whenever a new device is created. Only one profile is needed per device type. Some vendors provide pre-written device profiles for their devices. In this tutorial custom device templates will be used.

Sensor cluster

This device will be created manually to showcase how to use the EdgeX Foundry REST APIs. Scripts for doing this are also available and will be showcased later in the document.

The sensor cluster, which will be generating temperature and humidity data, will be created using Postman with the following steps:

- Create value descriptors
- Upload the device profile
- Create the device

Each step will include the same IP address - that of the host, and a port number. The port number determines which microservice is targeted with each command. For example:
- 48080: edgex-core-data
- 48081: edgex-core-metadata
- 48082: edgex-core-command
- etc.

1. Create value descriptors

   Value descriptors are what they sound like. They describe a value. They tell EdgeX what format the data comes in and what to label the data with. In this case value descriptors are created for temperature and humidity values respectively

   Open Postman and use the following values:
   Method: POST
   URI: http://<edgex ip>:48080/api/v1/valuedescriptor
   Payload settings: Set Body to "raw" and "JSON"
   Payload data:

```
{
    "name": "humidity",
    "description": "Ambient humidity in percent",
    "min": "0",
    "max": "100",
    "type": "Int64",
    "uomLabel": "humidity",
    "defaultValue": "0",
    "formatting": "%s",
    "labels": [
        "environment",
        "humidity"
    ]
}
```

   Tip: Use the very excellent feature "Beautify" on the "Body" tab in Postman to fix any indentation issues with the JSON payload.

   Watch for the return code of "**200 OK**" and the ID of the newly created value descriptor. There is no need to make note of the ID.
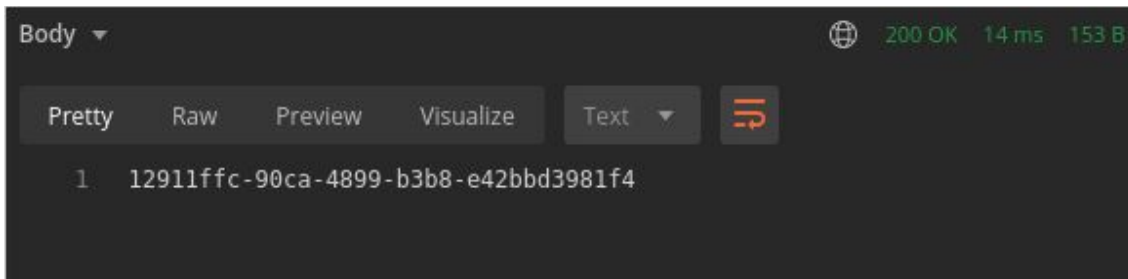
Update the body and issue the command again for temperature::
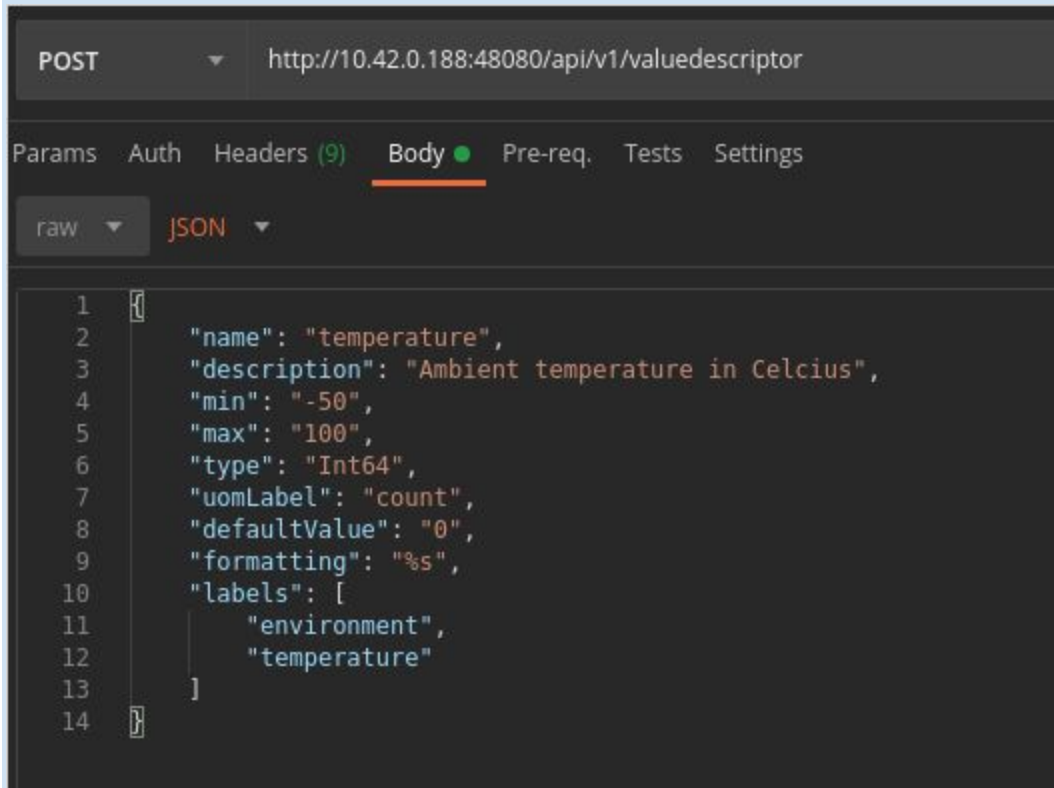Method: POST
URI: http://<edgex ip>:48080/api/v1/valuedescriptor
Payload settings: Set Body to "raw" and "JSON"
Payload data:

```
{
    "name": "temperature",
    "description": "Ambient temperature in Celsius",
    "min": "-50",
    "max": "100",
    "type": "Int64",
    "uomLabel": "temperature",
    "defaultValue": "0",
    "formatting": "%s",
    "labels": [
        "environment",
        "temperature"
    ]
}
```

Viewing value descriptors through Postman

2. Upload the device profile

Get a copy of the device profile from here:
https://raw.githubusercontent.com/jonas-werner/EdgeX_Tutorial/master/deviceCreation/sensorClusterDeviceProfile.yaml

In Postman use the following settings:
Method: POST
URI: http://<edgex ip>:48081/api/v1/deviceprofile/uploadfile
Payload settings: This part is a bit tricky:
- Set Body to "form-data"
- Hover over KEY and select "File"
- Select the yaml file: sensorClusterDeviceProfile.yaml
- In the KEY field, enter "file" as key
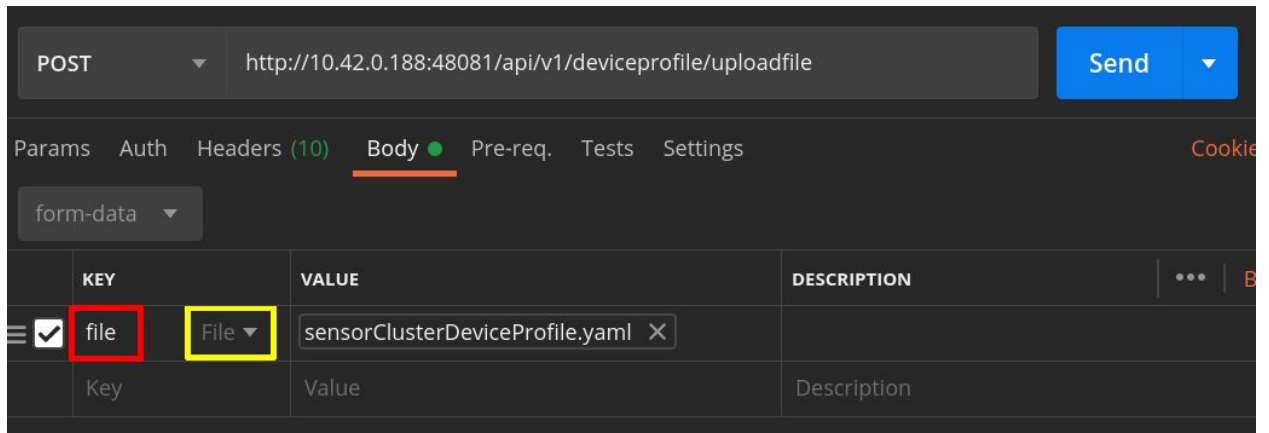
3. Create the device

Now EdgeX is finally ready to receive the device creation command in Postman as follows:

Two items are particularly important in this JSON body:
- The device service "edgex-device-rest" is used since this is a REST device.
- The profile name "SensorCluster" must match the name in the device profile yaml file uploaded in the previous step.

  Feel free to change values for description, location, labels, etc. if desired. However, the name (Temp_and_Humidity_sensor_cluster_01) will be referenced several times later and it's recommended to keep it at the default for now.

In Postman use the following settings:
Method: POST
URI: http://<edgex ip>:48081/api/v1/device
Payload settings: Set Body to "raw" and "JSON"
Payload data:

```
{
    "name": "Temp_and_Humidity_sensor_cluster_01",
    "description": "Raspberry Pi sensor cluster",
    "adminState": "unlocked",
    "operatingState": "enabled",
    "protocols": {
        "example": {
            "host": "dummy",
            "port": "1234",
            "unitID": "1"
```

```
            }
        },
        "labels": [
            "Humidity sensor",
            "Temperature sensor",
            "DHT11"
        ],
        "location": "Tokyo",
        "service": {
            "name": "edgex-device-rest"
        },
        "profile": {
            "name": "SensorCluster"
        }
}
```

If the above commands completed successfully a new sensor device will have been registered and EdgeX is ready to receive data from it.

Sending data to EdgeX Foundry

EdgeX is now ready to receive temperature and humidity data. To begin with, the functionality can be tested by posting individual data values using Postman. The next step after that is to use a Python script to simulate data values continuously. Finally, for those who wish to do so, a Raspberry Pi can be used to pull real values from a DHT humidity / temperature sensor and send these to EdgeX every few seconds.

The event counter

When data is sent to EdgeX it's registered as an event. It's possible to view the current event count with a browser (or cURL or Postman, etc.) here:
http://<edgex ip>:48080/api/v1/event/count

The page doesn't refresh by itself, so use F5 or the refresh button in the browser to view the latest event count during these examples.

Sending data with Postman

Sending individual data points with Postman is easy:

In Postman use the following settings to send a temperature value:
Method: POST
URI: http://<edgex ip>:49986/api/v1/resource/Temp_and_Hum
                           idity_sensor_cluster_01/temperature
Payload settings: Set Body to "raw" and "text"
Payload data: 23 (any integer value will do)

> Line is wrapped. Copy + paste to get rid of newline

Note:
  ● After submitting the data in Postman, look at the event count in the browser. Has it
    changed?
  ● Modify the URI to send a humidity value
  ● Extra points: Create another device (sensor_cluster_02 for example) and modify the URI
    to send data to that device instead
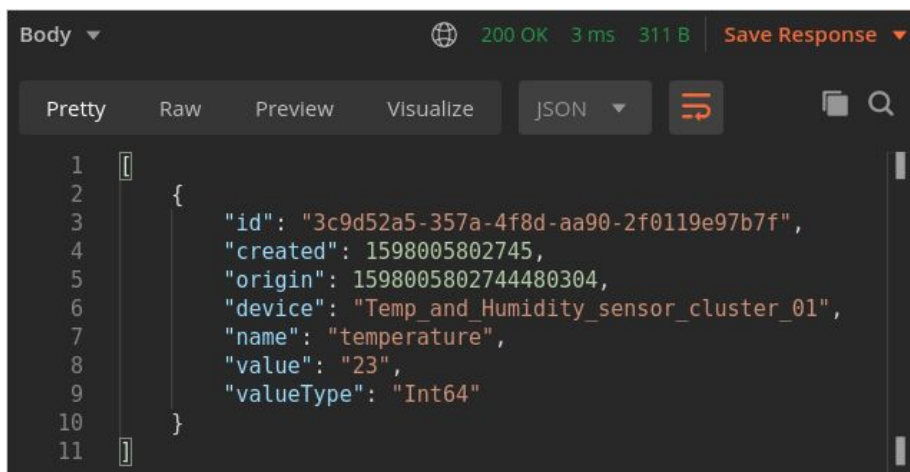
View the data

Use Postman to view the data stored in the EdgeX Foundry Redis DB as follows

In Postman use the following settings to view the temperature value:
Method: GET
URI: http://<edgex ip>:48080/api/v1/reading

The return data should be similar to the following:

```
Body ▼                              ⊕  200 OK  3 ms  311 B    Save Response ▼

 Pretty    Raw    Preview    Visualize    JSON  ▼   ⇥          📋  Q

  1  [
  2      {
  3          "id": "3c9d52a5-357a-4f8d-aa90-2f0119e97b7f",
  4          "created": 1598005802745,
  5          "origin": 1598005802744480304,
  6          "device": "Temp_and_Humidity_sensor_cluster_01",
  7          "name": "temperature",
  8          "value": "23",
  9          "valueType": "Int64"
 10      }
 11  ]
```

Generate sensor data with Python

It's possible to use a simple Python script to generate simulated sensor data continuously.

1. On the Linux VM, clone the Git repository for this tutorial:
   ```
   git clone https://github.com/jonas-werner/EdgeX_Tutorial.git
   ```
2. Enter the directory containing the data simulation script
   ```
   cd EdgeX_Tutorial/sensorDataGeneration/
   ```

   The Python module "requests" need to be installed to run the script. It is advisable to install modules in a separate virtual Python environment. Fortunately it's very quick to create one:

3. Install python3-venv
   ```
   sudo apt install python3-venv -y
   ```
4. Create a new virtual environment called simply "venv"
   ```
   python3 -m venv venv
   ```
5. Enter the virtual environment
   ```
   . ./venv/bin/activate
   ```
   or
   ```
   source ./venv/bin/activate
   ```
   Note that the terminal is now prefixed with the name of the virtual environment
6. Verify that the Python executable used is the one located in the virtual environment
   ```
   which python3
   ```

```
vagrant@edgex-tutorial:~/EdgeX_Tutorial/sensorDataGeneration$
vagrant@edgex-tutorial:~/EdgeX_Tutorial/sensorDataGeneration$ ls -l
total 12
-rw-rw-r-- 1 vagrant vagrant 1098 Aug 21 10:45 genSensorData.py
-rw-rw-r-- 1 vagrant vagrant  767 Aug 21 10:45 rpiPutTempHum.py
drwxrwxr-x 6 vagrant vagrant 4096 Aug 21 10:50 venv
vagrant@edgex-tutorial:~/EdgeX_Tutorial/sensorDataGeneration$
vagrant@edgex-tutorial:~/EdgeX_Tutorial/sensorDataGeneration$ . ./venv/bin/activate
(venv) vagrant@edgex-tutorial:~/EdgeX_Tutorial/sensorDataGeneration$
(venv) vagrant@edgex-tutorial:~/EdgeX_Tutorial/sensorDataGeneration$
(venv) vagrant@edgex-tutorial:~/EdgeX_Tutorial/sensorDataGeneration$ which python3
/home/vagrant/EdgeX_Tutorial/sensorDataGeneration/venv/bin/python3
(venv) vagrant@edgex-tutorial:~/EdgeX_Tutorial/sensorDataGeneration$ 
```

7. Install the requests module using pip
   ```
   pip install requests
   ```
8. If executing the script on any other host than the EdgeX Foundry VM, edit the file and change 127.0.0.1 to the IP address of the VM where EdgeX Foundry is installed
9. Run the script
   ```
   python3 ./genSensorData.py
   ```

10. To exit the script, use CTRL+C
11. To exit the virtual environment, simply type deactivate

    `deactivate`
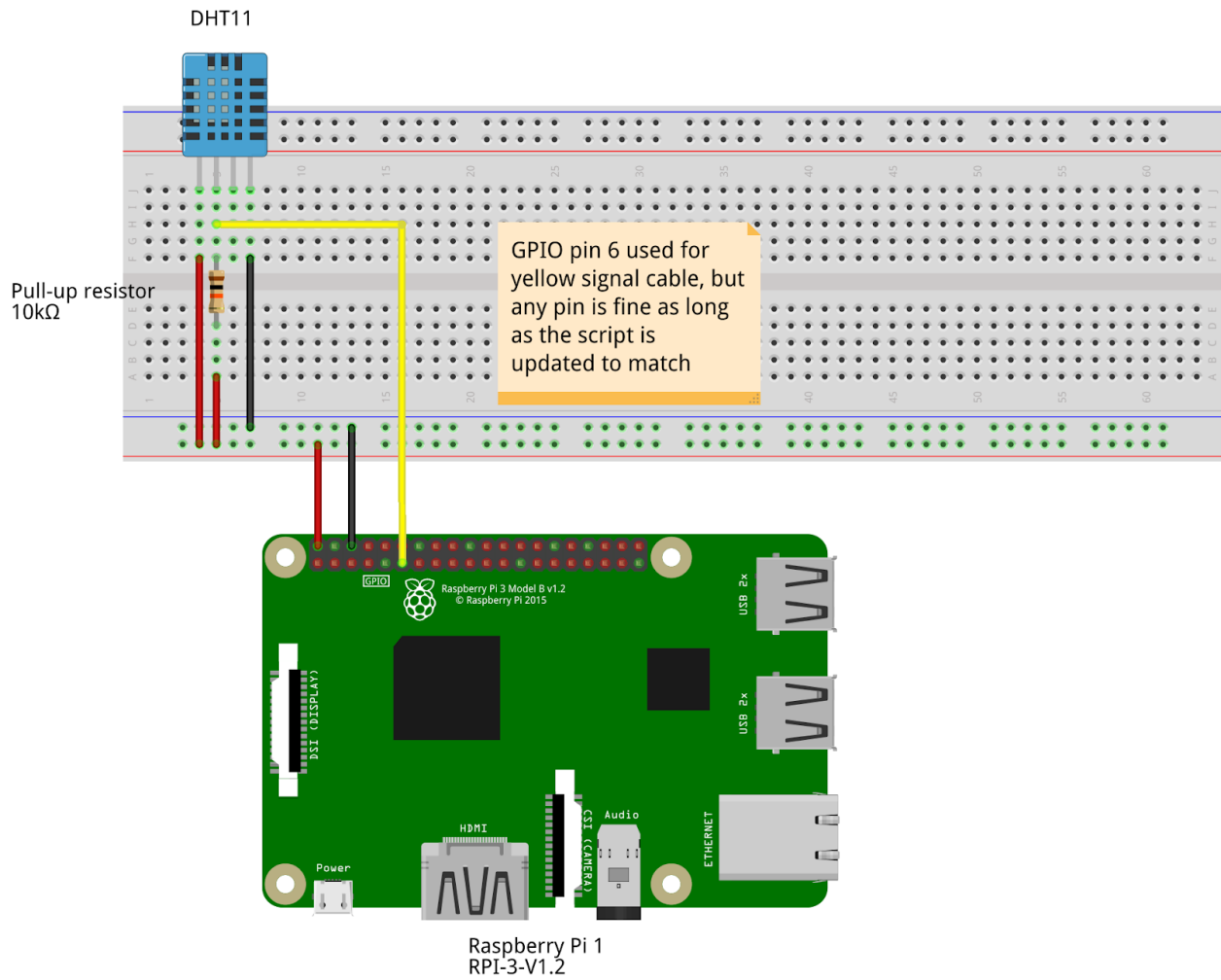
Using a DHT sensor on a Raspberry Pi

Optionally, to use real data from a real sensor, it is easy to use a Raspberry Pi and a DHT11 temperature / humidity sensor to pull in real world values and then push them to EdgeX Foundry using REST.

Prerequisites

- Raspberry Pi (version 3b+ used in examples below)
- DHT sensor (DHT11 used in examples)
- Raspbian / Ubuntu (Stretch used in examples)
- SSH access enabled
- IP reachability between RPi and EdgeX Foundry (bridged interface for EdgeX VM used in examples)

Sensor wiring diagram

Example below of DHT11 connected to GPIO of a Raspberry Pi 3b+

DHT11

Pull-up resistor
10kΩ

GPIO pin 6 used for
yellow signal cable, but
any pin is fine as long
as the script is
updated to match

Raspberry Pi 1
RPI-3-V1.2

fritzing

Network diagram

At this step data is captured from the DHT11 using the Adafruit_DHT library, converted into
JSON and sent via REST to the EdgeX Foundry VM. Data export with MQTT will be added in
the next section
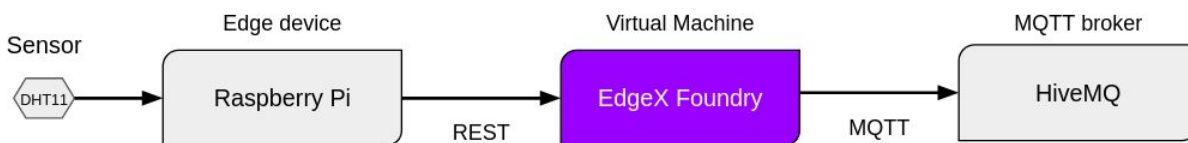
Setup instructions

1. Download Python requirements file and script from GitHub:
   - ```
     wget
     https://raw.githubusercontent.com/jonas-werner/EdgeX_Tutor
     ial/master/sensorDataGeneration/requirements.txt
     ```
   - ```
     wget
     https://raw.githubusercontent.com/jonas-werner/EdgeX_Tutor
     ial/master/sensorDataGeneration/rpiPutTempHum.py
     ```
2. Install Python3 venv for the virtual environment
   ```
   apt install python3-venv -y
   ```
3. Create virtual environment (name: "venv")
   ```
   python3 -m venv venv
   ```
4. Enter virtual environment
   ```
   . ./venv/bin/activate
   ```
5. Install Python modules
   ```
   pip install -r requirements.txt
   ```

```
pi@sensorcluster:~/tutorial $ python3 -m venv venv
pi@sensorcluster:~/tutorial $ ls -l
total 12
-rw-r--r-- 1 pi pi  120 Aug 24 01:46 requirements.txt
-rw-r--r-- 1 pi pi  793 Aug 24 01:46 rpiPutTempHum.py
drwxr-xr-x 6 pi pi 4096 Aug 24 01:47 venv
pi@sensorcluster:~/tutorial $ . ./venv/bin/activate
(venv) pi@sensorcluster:~/tutorial $
(venv) pi@sensorcluster:~/tutorial $ pip install -r requirements.txt
Collecting Adafruit-DHT==1.4.0 (from -r requirements.txt (line 1))
  Using cached https://www.piwheels.org/simple/adafruit-dht/Adafruit_DHT-1.4.0-cp35-cp35m-linux
Collecting certifi==2020.6.20 (from -r requirements.txt (line 2))
  Using cached https://files.pythonhosted.org/packages/5e/c4/6c4fe722df5343c33226f0b4e0bb042e4d
.6.20-py2.py3-none-any.whl
```

6. Edit the script to update the EdgeX Foundry IP address on the following line:
   ```
   edgexip = "<edgex ip>"
   ```
7. Edit the script to match sensor (DHT11 or 22) and GPIO pin used on the following line:
   ```
   rawHum, rawTmp = Adafruit_DHT.read_retry(11, 6)
   ```
   Note: In this example DHT11 and GPIO pin 6 is used
8. Run the script
   ```
   python ./putTempHumidity.py
   ```

```
(venv) pi@sensorcluster:~/tutorial $ python ./putTempHumidity.py
Temp: 24.0°C, humidity: 88.0%
Temp: 24.0°C, humidity: 89.0%
Temp: 24.0°C, humidity: 89.0%
Temp: 24.0°C, humidity: 89.0%
Temp: 24.0°C, humidity: 92.0%
Temp: 24.0°C, humidity: 95.0%
Temp: 26.0°C, humidity: 95.0%
Temp: 26.0°C, humidity: 91.0%
Temp: 26.0°C, humidity: 89.0%
Temp: 26.0°C, humidity: 87.0%
Temp: 26.0°C, humidity: 86.0%
Temp: 26.0°C, humidity: 85.0%
```

Data is now sent to EdgeX Foundry and stored short-term in the Redis DB. Since data isn't meant to be stored at the edge device for long, it's important to configure the data export settings. That way the data will be sent via EdgeX to some centralized location for storage and/or processing. How to configure data export is covered in the next chapter.

Export data stream

Data export to an external source, like an MQTT topic, AWS or Azure, is generally done using the Application Service. This service can be configured by adding options to the docker-compose.yml file. Examples are posted to the EdgeX Foundry documentation page: https://docs.edgexfoundry.org/1.2/microservices/application/AppServiceConfigurable/

The data can also be exported selectively by using the Rules Engine (Kuiper). In this case an SQL statement can be used to pick up on certain data and export it as desired. Both methods will be shown.

MQTT export using the Application Service

In this example a new docker-compose.yml file is used to get up and running quickly. It has been pre-configured with settings for exporting data to a public MQTT broker: HiveMQ. The only change needed is to update the topic to a unique name so that it can be easily subscribed to using the HiveMQ web interface.

1. On the EdgeX Foundry VM, enter the "geneva" folder
   `cd geneva/`
2. Stop EdgeX Foundry (if it is running)
   `docker-compose stop`

3. Download the new docker-compose file from GitHub
```
wget
https://raw.githubusercontent.com/jonas-werner/EdgeX_Tutorial/m
aster/docker-compose_files/docker-compose_step2.yml
```

4. Copy the new docker-compose file to "docker-compose.yml"
   Note: Make a backup of the original if you want to keep your modifications
```
cp docker-compose_step2.yml docker-compose.yml
```

5. Edit the docker-compose.yml file and enter a unique MQTT topic ID instead of the entry "YOUR-UNIQUE-TOPIC". Anything is fine as long as it's unique and memorable. Avoid spaces and special characters. Don't forget the quotation marks.

6. Start EdgeX Foundry
```
docker-compose up -d
```

   Note: If Portainer and the EdgeX Go UI was added previously, use the following to clean up those containers (no actual orphans will be affected by this command):
```
docker-compose up -d --remove-orphans
```

7. Open a browser and go to: http://www.hivemq.com/demos/websocket-client/
8. Click on Connect (all the default values are fine)
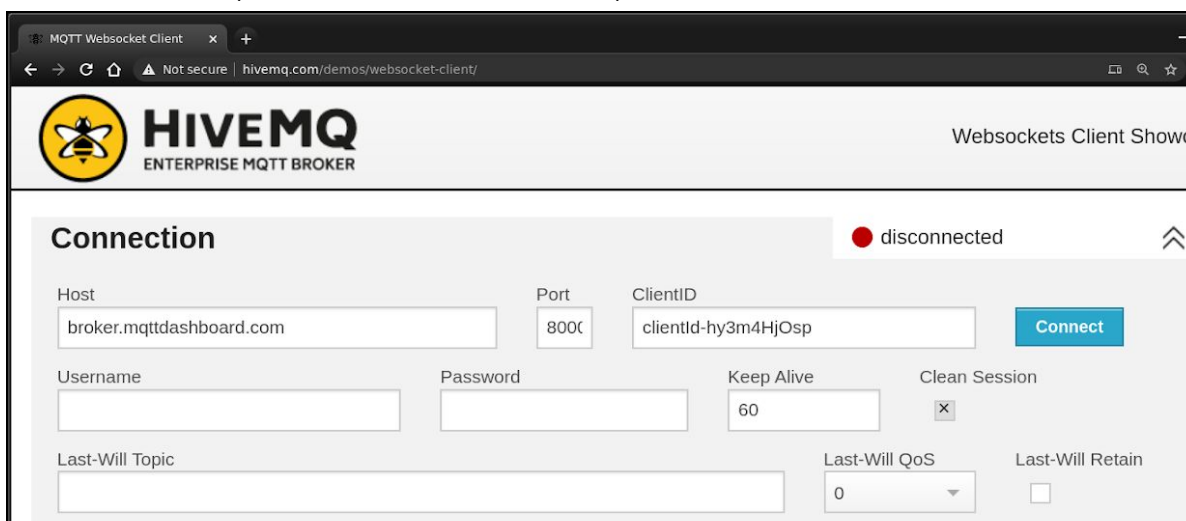
9. Click on "Add New Topic Subscription" and enter the name of your MQTT topic exactly as it is listed in the docker-compose.yml file.
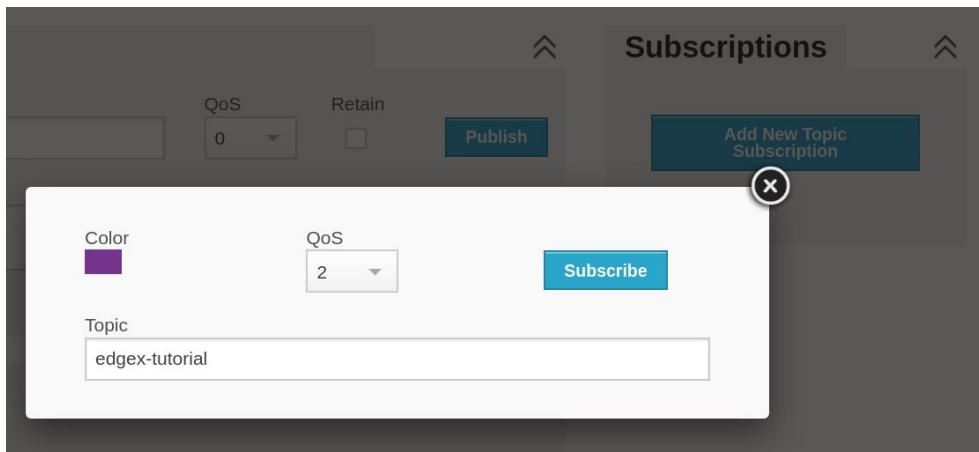
10. Click "Subscribe"

11. All data sent to EdgeX Foundry will now be processed and sent to this MQTT topic. Generate some data and see it appear in the HiveMQ web console:
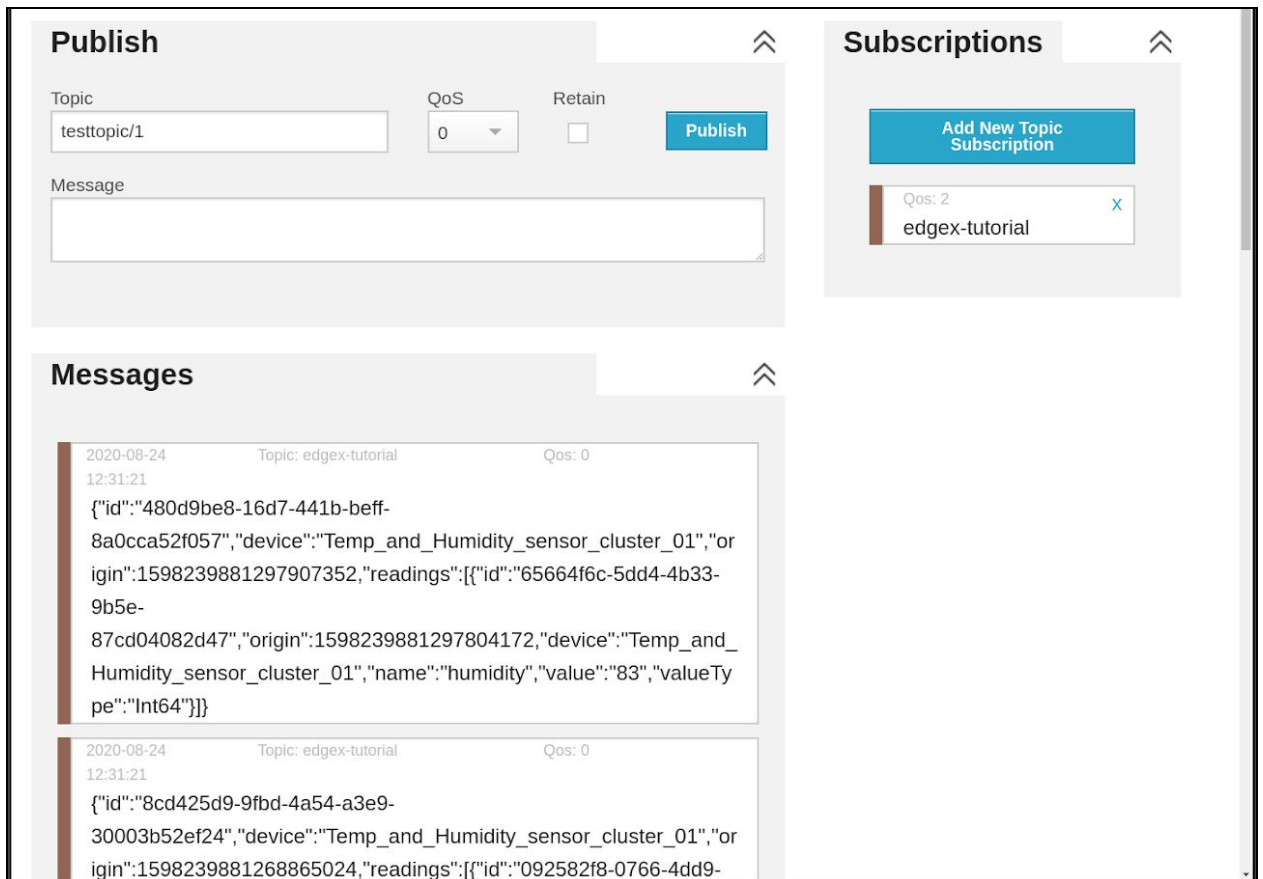
Congratulations - you're now ingesting, processing and exporting data while converting between REST and MQTT protocols.

MQTT export using the Rules Engine

Data can also be exported more selectively by using Kuiper - the EdgeX rules engine. Kuiper uses the concept of streams. Rules will link with a stream to execute actions based on SQL statements.

Postman will be used to create a stream and then a rule linking with that stream. The rule will be configured to capture all data from the stream and export it using MQTT to a HiveMQ topic. As per the screenshot below, Kuiper runs on port 48075. This can be verified by entering the "geneva" folder and executing:

```
docker-compose ps | grep 48075
```


1. Creating a Kuiper stream with Postman

   Method: POST
   URI: http://<edgex ip>:48075/streams
   Payload settings: Set Body to "raw" and "JSON"
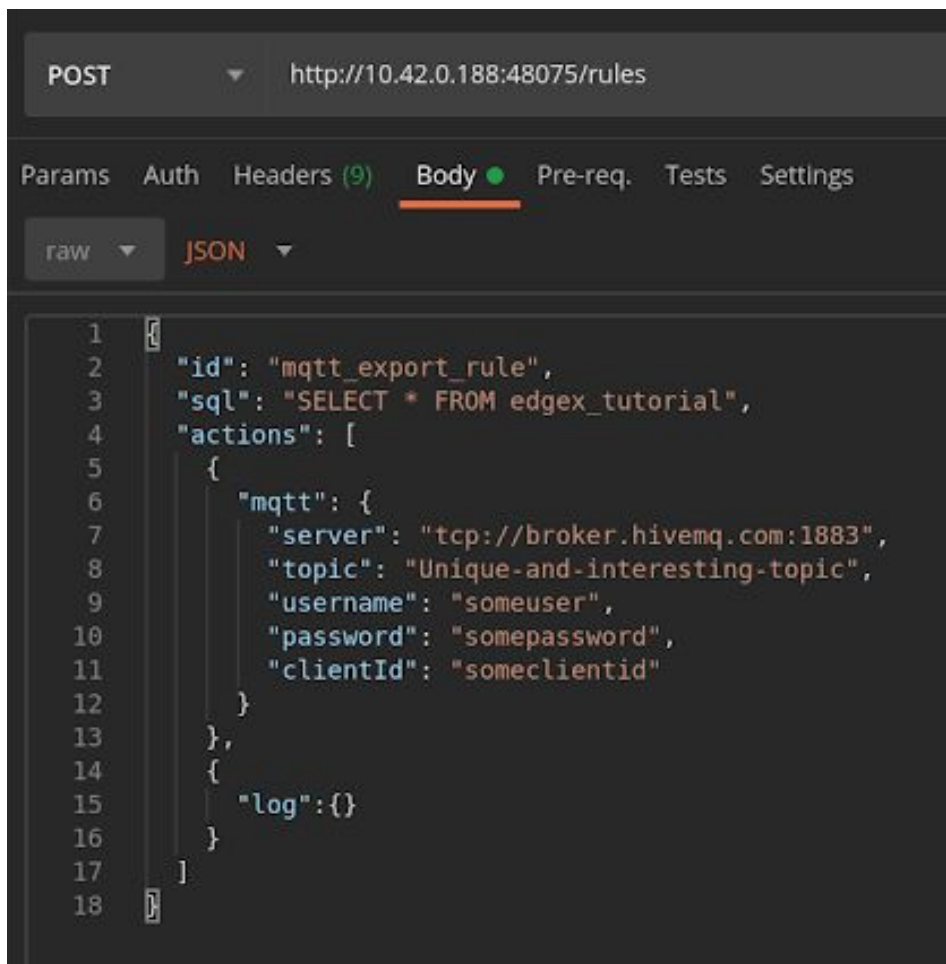   Payload data:

```
{
    "sql": "create stream edgex_tutorial() WITH
(FORMAT=\"JSON\", TYPE=\"edgex\")"
}
```

2. Creating a Kuiper rule with Postman

Method: POST
URI: http://<edgex ip>:48075/rules
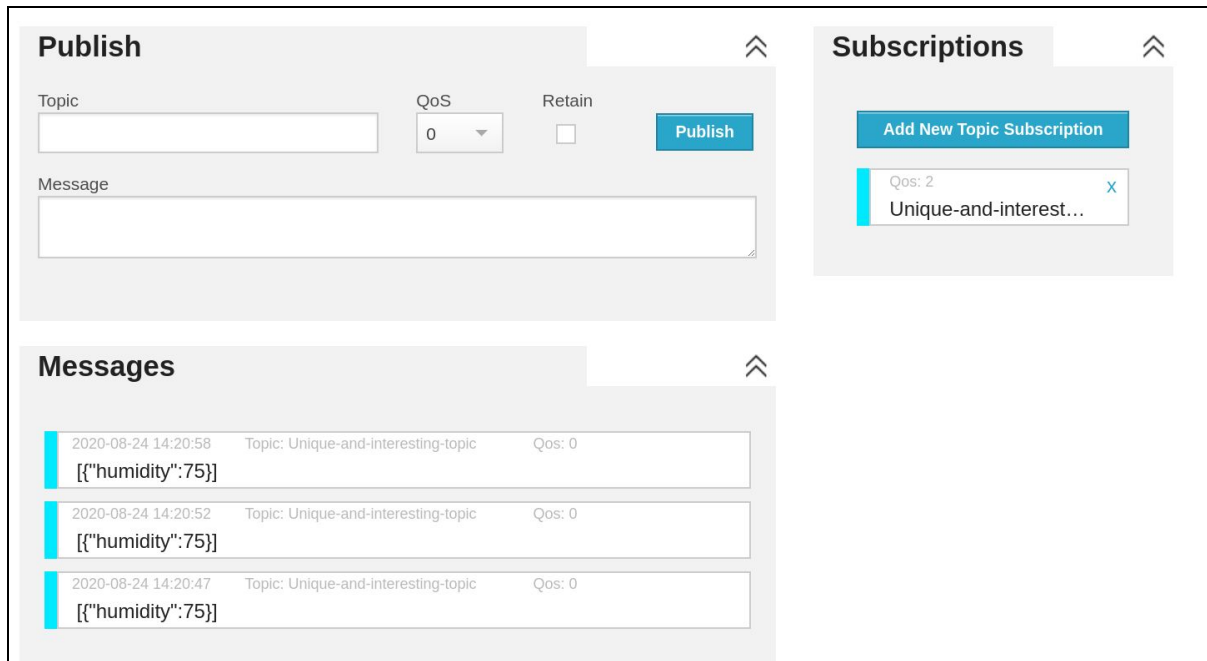Payload settings: Set Body to "raw" and "JSON"
Payload data:

```json
{
  "id": "mqtt_export_rule",
  "sql": "SELECT * FROM mqtt_export",
  "actions": [
    {
      "mqtt": {
        "server": "tcp://broker.hivemq.com:1883",
        "topic": "EdgeXFoundryMQTT_01",
        "username": "someuser",
        "password": "somepassword",
        "clientId": "someclientid"
      }
    },
    {
      "log":{}
    }
  ]
}
```

3. Verifying that HiveMQ is receiving the data
   Visit http://www.hivemq.com/demos/websocket-client/
   Click "Connect" (default values are fine)

4. Add the topic subscription and send some data to EdgeX Foundry

Note: Since this is a public test service from HiveMQ, QoS isn't guaranteed. It can sometimes take a little while for messages to appear in the web UI. Alternatively it's possible to set up a private MQTT broker using Mosquitto or of course to use one of the paid offerings available from a variety of companies.

Sending commands

A sometimes overlooked feature of EdgeX Foundry is its ability to send commands to devices. In this section commands will be demonstrated using a test application. The application runs in a container and has a web service which can be updated over a REST API. This can be used by the app to receive commands from EdgeX Foundry and execute changes to a web interface viewable through a browser.

Note: The current REST device service doesn't yet support commands. Therefore we'll be using a legacy function of EdgeX to get around this limitation by creating a dummy device service. It's a bit of a hack, but it works ¯_(ツ)_/¯

This section is broken up into the following steps:
● Building and running the test app container
● Registering the app as a new device
● Issuing commands via EdgeX
● Creating a rule to execute a command

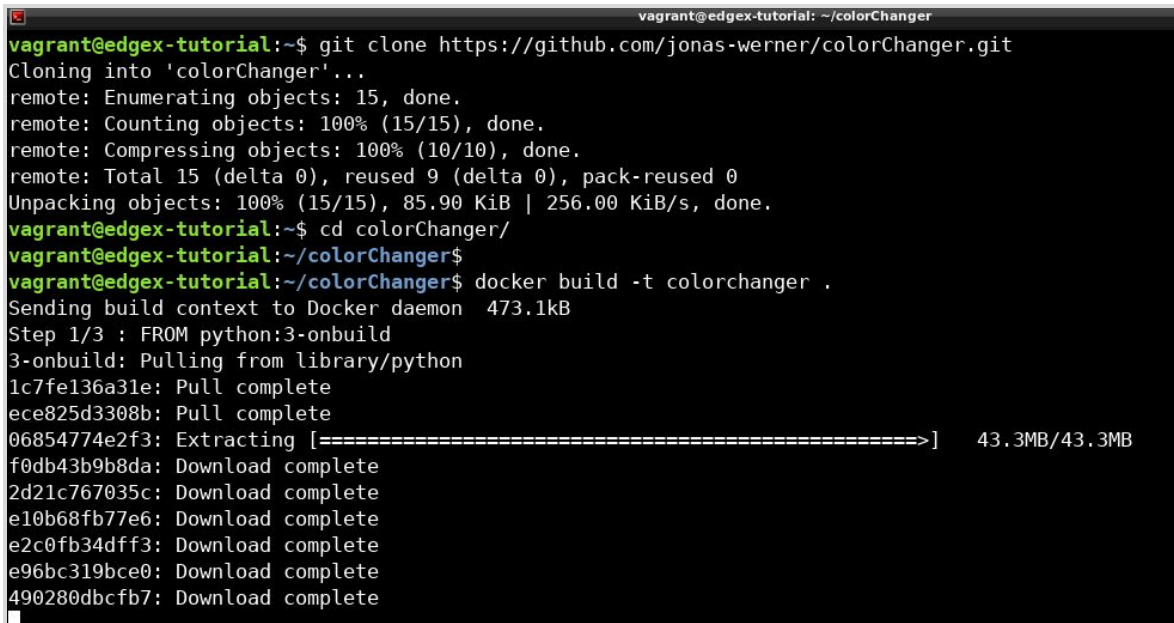## Building and running the test app container

1. On the EdgeX Foundry VM, clone the repository for the test app container

    ```
    git clone https://github.com/jonas-werner/colorChanger.git
    ```

2. Enter the directory and build the container

    ```
    cd colorChanger/
    docker build -t colorChanger .
    ```

```
vagrant@edgex-tutorial: ~/colorChanger
vagrant@edgex-tutorial:~$ git clone https://github.com/jonas-werner/colorChanger.git
Cloning into 'colorChanger'...
remote: Enumerating objects: 15, done.
remote: Counting objects: 100% (15/15), done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 15 (delta 0), reused 9 (delta 0), pack-reused 0
Unpacking objects: 100% (15/15), 85.90 KiB | 256.00 KiB/s, done.
vagrant@edgex-tutorial:~$ cd colorChanger/
vagrant@edgex-tutorial:~/colorChanger$
vagrant@edgex-tutorial:~/colorChanger$ docker build -t colorchanger .
Sending build context to Docker daemon   473.1kB
Step 1/3 : FROM python:3-onbuild
3-onbuild: Pulling from library/python
1c7fe136a31e: Pull complete
ece825d3308b: Pull complete
06854774e2f3: Extracting [=================================================>]   43.3MB/43.3MB
f0db43b9b8da: Download complete
2d21c767035c: Download complete
e10b68fb77e6: Download complete
e2c0fb34dff3: Download complete
e96bc319bce0: Download complete
490280dbcfb7: Download complete
```

3. Run the container

    ```
    docker run -d -p 5000:5000 --name colorchanger
    colorchanger:latest
    ```

4. Verify that the web interface of the container is accessible by using a web browser:
    http://<edgex ip>:5000
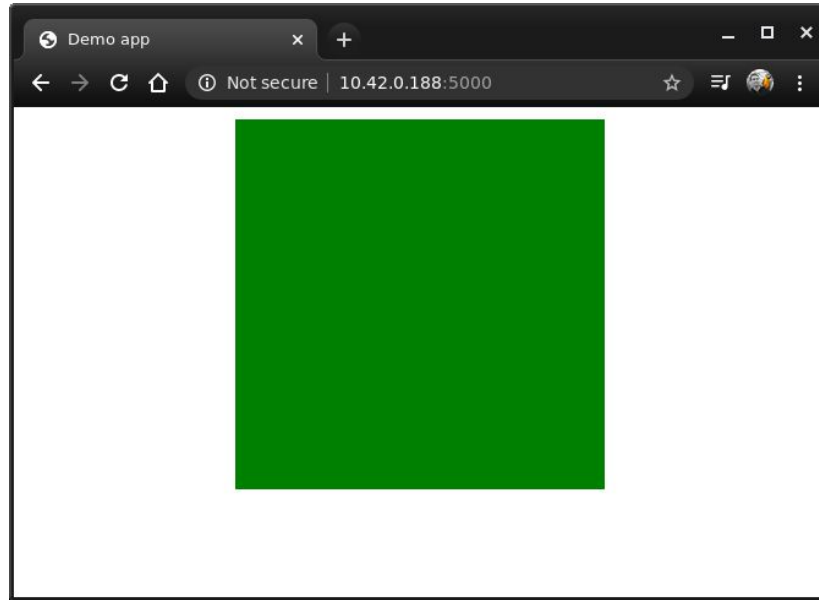
The default web interface of the test app. A green square. Very exciting.

5. Use Postman to test the REST API
    Method: PUT
    URI: http://<edgex ip>:5000/api/v1/device/edgexTutorial/changeColor
    Payload settings: Set Body to "raw" and "JSON"
    Payload data:
    ```
    {
        "color": "navy"
    }
    ```
6. The web page should automatically change color

By using this it's possible to test sending REST commands and verify functionality

Registering the app as a new device
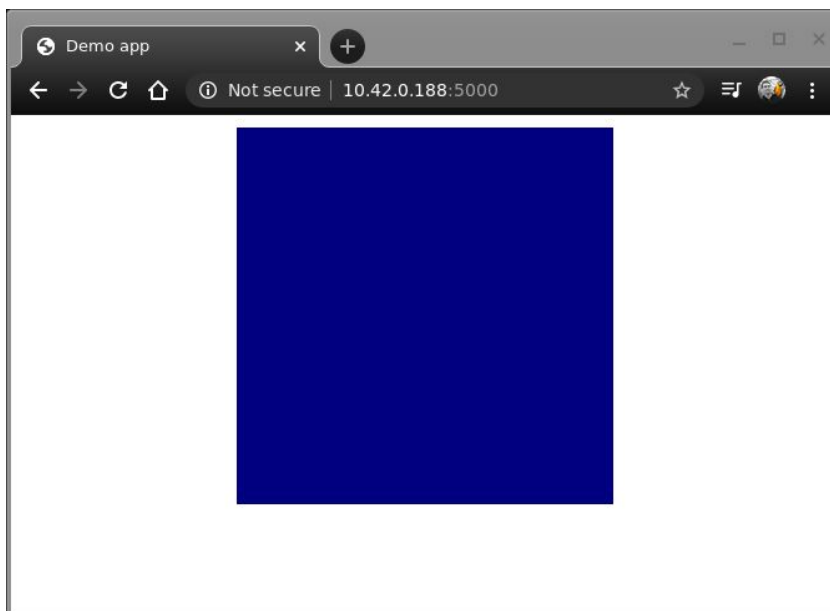
EdgeX Foundry needs to know about the test app and how to interact with it. To accomplish this a new device template is used together with a Python script to run all commands required to register the app in one go.

The script will create the below entries and link them together. It includes a few more steps compared with the sensor cluster created previously, but since the script does the work it only takes a fraction of a second to complete.
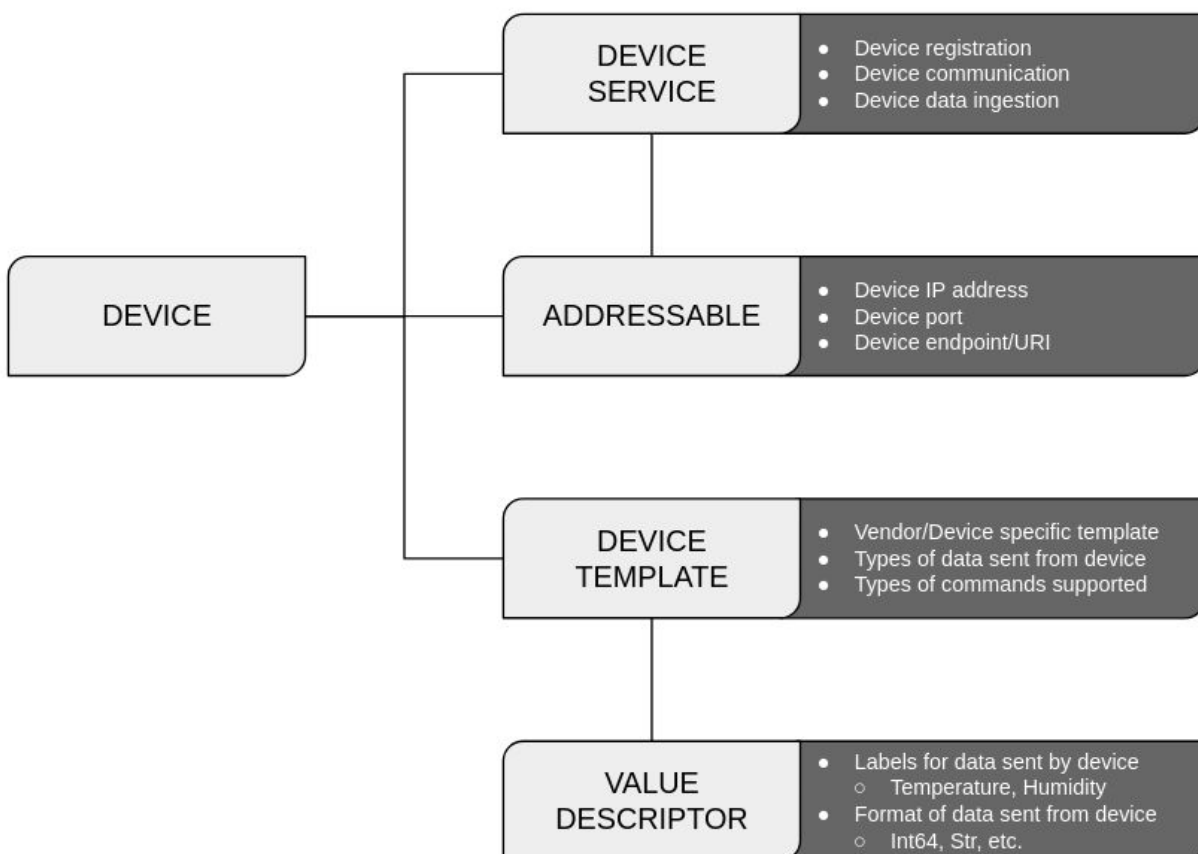
Relation between entities created by the Python script

Access the EdgeX Foundry VM.
1. If not cloned already, on the Linux VM clone the Git repository for this tutorial:
   ```
   git clone https://github.com/jonas-werner/EdgeX_Tutorial.git
   ```
2. Enter the directory containing the device creation scripts:
   ```
   cd EdgeX_Tutorial/deviceCreation
   ```

The "requests" and "requests-toolbelt" Python modules need to be installed to run the script. Create a virtual Python environment. If a virtual environment has already been created since before, please skip to step 5 and enter the environment.

3. Install python3-venv
```
sudo apt install python3-venv -y
```
4. Create a new virtual environment called simply "venv"
```
python3 -m venv venv
```
5. Enter the virtual environment
```
. ./venv/bin/activate
```
or
```
source ./venv/bin/activate
```
Note that the terminal is now prefixed with the name of the virtual environment
6. Verify that the Python executable used is the one located in the virtual environment
```
which python3
```

7. Install the requests module using pip
```
pip install requests
```
8. Install the requests-toolbelt module using pip
```
pip install requests_toolbelt
```
9. Run the script
   Note:
   ● The "-ip" entry refers to the EdgeX Foundry host IP.
   ● The "-devip" entry refers to the IP of the host running the test app.
   ● In this example it is assumed the test app is running on the same host as EdgeX Foundry. As such, use the same IP for both of them below:
```
python ./createRESTDevice.py -ip <edgex ip> -devip <edgex ip>
```

```
(venv) vagrant@edgex-tutorial:~/EdgeX_Tutorial/deviceCreation$ ll
total 36
drwxrwxr-x 3 vagrant vagrant 4096 Aug 24 10:43 ./
drwxrwxr-x 6 vagrant vagrant 4096 Aug 24 10:43 ../
-rw-rw-r-- 1 vagrant vagrant 6376 Aug 24 10:43 createRESTDevice.py
-rw-rw-r-- 1 vagrant vagrant 5970 Aug 24 10:43 createSensorCluster.py
-rw-rw-r-- 1 vagrant vagrant 1133 Aug 24 10:43 RESTDeviceProfile.yaml
-rw-rw-r-- 1 vagrant vagrant  698 Aug 24 10:43 sensorClusterDeviceProfile.yaml
drwxrwxr-x 6 vagrant vagrant 4096 Aug 24 10:43 venv/
(venv) vagrant@edgex-tutorial:~/EdgeX_Tutorial/deviceCreation$
(venv) vagrant@edgex-tutorial:~/EdgeX_Tutorial/deviceCreation$ python ./createRESTDevice.py -ip 10.42.0.188 -devip 10.42.0.188
Result for createAddressables: <Response [200]> - Message: c520b134-b377-4f17-bf24-b5a3e7f25f6a
Result for createValueDescriptors #1: <Response [200]> - Message: 66775f02-edaf-4a28-8b79-7e68de878be2
Result of uploading device profile: <Response [200]> with message b3104968-196f-4339-b655-f7c69f5afee9
Result for createDeviceService: <Response [200]> - Message: 8a4f0abe-ba69-41e1-971a-621ab5b8f9d6
Result for addNewDevice: <Response [200]> - Message: 01671ea6-b75a-4c7c-8afc-8a660d7dd30c
(venv) vagrant@edgex-tutorial:~/EdgeX_Tutorial/deviceCreation$
```

The script should return a "[200]" response for each REST call to EdgeX Foundry
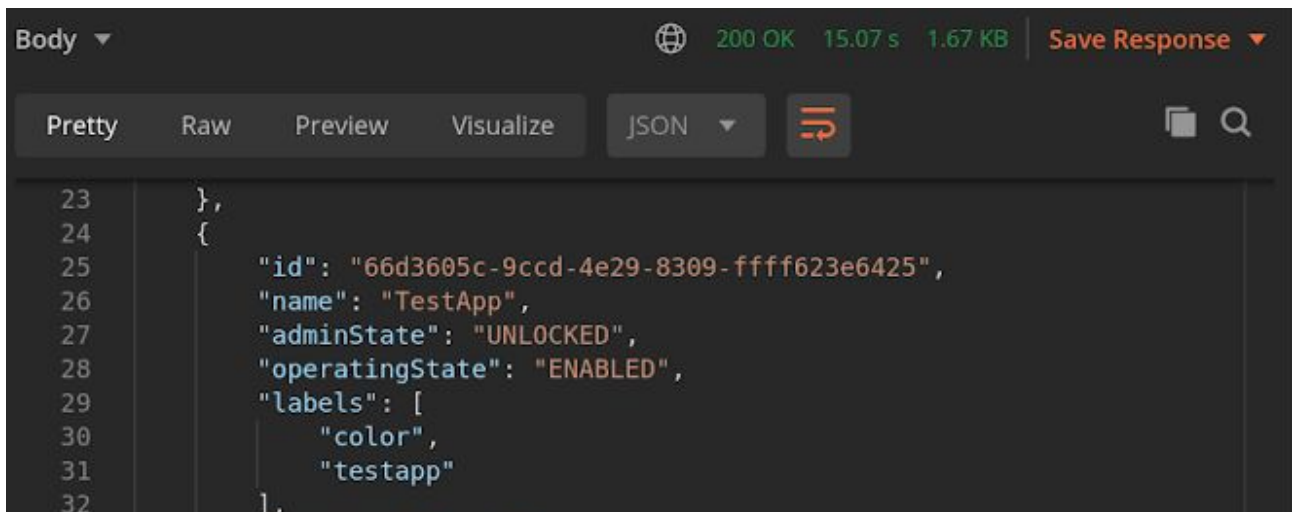
Issuing commands via EdgeX

With the new device created it's now possible to view it in EdgeX Foundry using Postman. It is also possible to use the EdgeX Foundry REST API to communicate with the test application. EdgeX will automatically have created a unique REST endpoint for this new device. We can use it to send commands without having to talk to the device directly, but via EdgeX Foundry.

In this case a REST call to EdgeX will be translated to a REST call to the device, so no protocol translation. However, if the device was using a different protocol the translation would take place.

1. View the new TestApp device details using Postman
   Method: GET
   URI: http://<edgex ip>:48082/api/v1/device

   The device "TestApp" is now visible in the list of registered devices

   Scrolling down reveals a "commands" section which thanks to the Device Profile used has been equipped with both "get" and "put" commands with IDs unique to this device.

   Find the entry for "url" under "put".
   Note: "get" will have it's own url, but it's not used in this tutorial.

```
    }
"put": {
    "path": "/api/v1/device/{deviceId}/changeColor",
    "responses": [
        {
            "code": "201",
            "description": "set the color"
        },
        {
            "code": "503",
            "description": "service unavailable"
        }
    ],
    "url": "http://edgex-core-command:48082/api/v1/
        device/01671ea6-b75a-4c7c-8afc-8a660d7dd30c/
        command/8948b979-cebb-4727-b898-03f76268a136"
}
```

Clicking the "url" for "put" will open a new tab in Postman

2. Click the URL for "put":
    A new Postman tab is opened.
    Replace "edgex-core-command" with the IP of EdgeX Foundry

    Method: PUT
    URI: populated by clicking the "url" link above (unique for each device)
    Payload settings: Set Body to "raw" and "JSON"
    Payload data:
```
{
    "color": "orange"
}
```

```
PUT   ▼   http://10.42.0.188:48082/api/v1/device/66d3605c-9ccd-4e29-8309-ffff623e6425/command   Send   ▼

Body ▼                                        •••   Body ▼        ⊕  200 OK  7 ms  132 B │ Save

raw ▼   JSON ▼                         Beautify       Pretty ▼   HTML ▼   ⇥

  1  {                                                   1   Command accepted
  2      "color": "orange"
  3  }
```
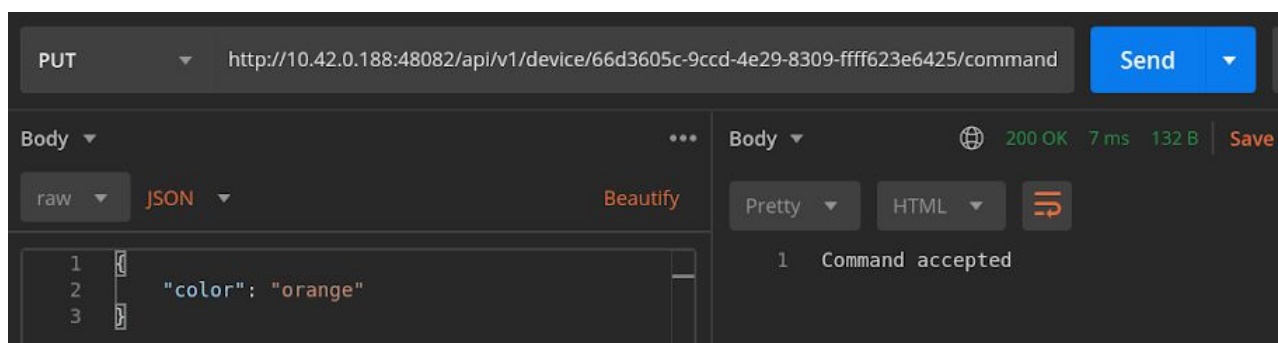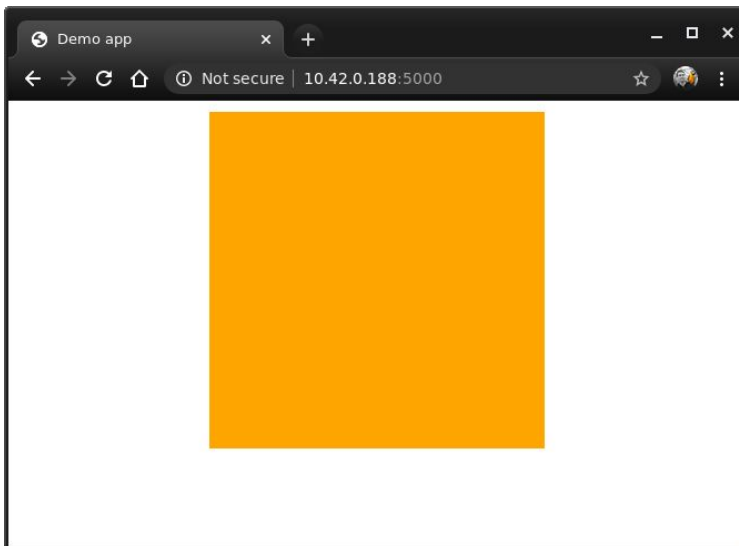
3. View the TestApp through a browser
    http://<edgex ip>:5000

Did it change color? Note that in this case we're issuing the REST call to EdgeX Foundry, not to the test app itself. EdgeX receives, converts and re-issues the command

Creating a rule to execute commands automatically

It may be desirable to have a command executed automatically whenever a threshold is met based on sensor input. For example, take some pre-set action if the temperature goes over a certain limit. The Kuiper rules engine can help with that.

1. Create a new Kuiper stream with Postman
   Method: POST
   URI: http://<edgex ip>:48075/streams
   Payload settings: Set Body to "raw" and "JSON"
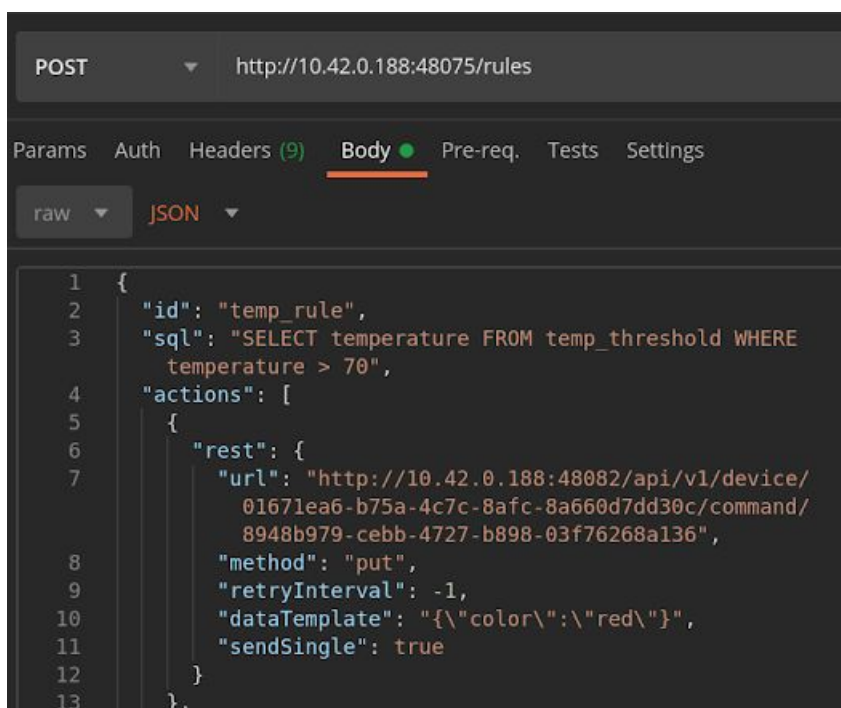   Payload data:

   ```
   {
       "sql": "create stream temp_threshold() WITH
   (FORMAT=\"JSON\", TYPE=\"edgex\")"
   }
   ```

2. Create a new Kuiper rule which links to the stream
   Method: POST
   URI: http://<edgex ip>:48075/rules
   Payload settings: Set Body to "raw" and "JSON"
   Payload data:

```json
{
  "id": "temp_rule",
  "sql": "SELECT temperature FROM temp_threshold WHERE temperature > 70",
  "actions": [
    {
      "rest": {
        "url": "<unique command for put>",
        "method": "put",
        "retryInterval": -1,
        "dataTemplate": "{\"color\":\"red\"}",
        "sendSingle": true
      }
    },
    {
      "log":{}
    }
  ]
}
```

Note: Copy and paste the command for "put" as listed for the TestApp device when querying EdgeX on http:<edgex ip>:48082/api/v1/device

Since this command is unique to each device and every installation of EdgeX Foundry it's necessary to copy and paste after searching for it. An example is shown below:
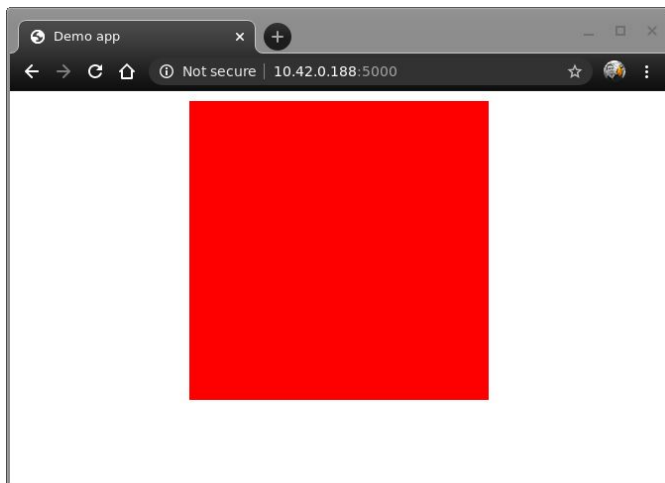
3. Push a temp value over 70 degrees with Postman to trigger the rule

Method: POST

URI: http://<edgex ip>:49986/api/v1/resource/Temp_a nd_Humidity_sensor_cluster_01/temperature

Payload settings: Set Body to "raw" and "text"

Payload data: 71 (any numeric value over 70 will do)

> Line is wrapped. Copy + paste to get rid of newline

The web app should switch to red

4. **Extra points:** Create another rule to change the color back if the temperature drops below 70 degrees. Flip between the different states by sending different temperature values.

Viewing container logs

There are more advanced logging features available but they're out of scope for this guide. However, it's easy to do basic troubleshooting by looking at the log output of the individual containers.

1. Access the EdgeX Foundry VM
2. List up the containers with docker-compose ps

```
docker-compose ps
```

3. Note the container names to the left
4. Use the docker logs command to view the log output:

```
docker logs <container name>
```

For example

```
docker logs -f edgex-core-data
```

5. To view output continuously (until CTRL+C is pressed), add the "-f" flag

```
docker logs -f <container name>
```

Bonus: Visualize data

While not part of EdgeX Foundry, it can be useful to know how to capture, save and visualize data. This section showcases one way of doing so by capturing data from an MQTT topic.

Four new containerized apps will be added to the host VM running EdgeX Foundry (they could be located anywhere though - even somewhere in the cloud).

The apps are:
- Mosquitto: A MQTT broker which can be run locally
- InfluxDB: A time series database perfect for capturing sensor data over time
- Grafana: A graphical dashboard which supports InfluxDB
- Messenger: A python script in a container which captures MQTT messages and enters them into the InfluxDB

Adding new containers

Execute all commands on the EdgeX Foundry VM

1. Download and run Mosquitto
   ```
   docker pull eclipse-mosquitto

   docker run --name mosquitto -d -p 1883:1883 -p 9001:9001
   eclipse-mosquitto
   ```

2. Download and run InfluxDB
   ```
   docker pull influxdb

   docker run \
        -d \
        --name influxdb \
        -p 8086:8086 \
        -e INFLUXDB_DB=sensordata \
        -e INFLUXDB_ADMIN_USER=root \
        -e INFLUXDB_ADMIN_PASSWORD=pass \
        -e INFLUXDB_HTTP_AUTH_ENABLED=true \
        influxdb
   ```

3. Download and run Grafana
   ```
   docker pull grafana/grafana
   ```

```
docker run -d --name=grafana -p 3000:3000 grafana/grafana
```

4. Update the IP settings in the "messenger" app
   If the EdgeX_Tutorial repository has been cloned from GitHub already, enter the
   directory holding the messenger app files:
   ```
   cd EdgeX_Tutorial/messenger
   ```

   Otherwise clone it from https://github.com/jonas-werner/EdgeX_Tutorial.git and enter the
   folder.

   Open the file "app.py' in an editor and replace "<edgex ip>" with the actual IP address of
   the VM the app will run on. Likely the EdgeX Foundry VM IP address in this case.
   Note: There are two entries! Don't use the loopback address (127.0.0.1) as the ports will
   clash with the mosquitto MQTT broker.

```
22    # on line 92 and 95 if required
23    broker_address  = "<edgex ip>"
24    topic           = "edgex-tutorial"
25    dbhost          = "<edgex ip>"
26    dbport          = 8086
27    dbuser          = "root"
28    dbpassword      = "pass"
29    dbname          = "sensordata"
30
```

5. Build the container for the messenger app
   ```
   docker build -t messenger .
   ```
   Note: Make sure to execute the "docker build" command in the
   "EdgeX_Tutorial/messenger" directory

6. Run the messenger app
   ```
   docker run -d --name messenger messenger:latest
   ```

Redirecting EdgeX to the local MQTT broker

There should now be four new containers running. To get EdgeX Foundry to send the MQTT
messages to the local Mosquitto MQTT broker instead of HiveMQ we need to edit the
docker-compose.yml file.

1. Stop EdgeX Foundry if it's running
   ```
   docker-compose down
   ```

2. Enter the "geneva" folder and edit the "docker-compose.yml" file
   Change the broker address from: "broker.hivemq.com" to the IP address of the host
   running the Mosquitto broker. In this case it's the IP of the host VM running EdgeX
   Foundry.

```
Binding_PublishTopic: events
# Added for MQTT export using app service
WRITABLE_PIPELINE_FUNCTIONS_MQTTSEND_ADDRESSABLE_ADDRESS: "10.42.0.188"
WRITABLE_PIPELINE_FUNCTIONS_MQTTSEND_ADDRESSABLE_PORT: 1883
WRITABLE_PIPELINE_FUNCTIONS_MQTTSEND_ADDRESSABLE_PROTOCOL: tcp
WRITABLE_PIPELINE_FUNCTIONS_MQTTSEND_ADDRESSABLE_TOPIC: "edgex-tutorial"
WRITABLE_PIPELINE_FUNCTIONS_MQTTSEND_PARAMETERS_AUTORECONNECT: "true"
WRITABLE_PIPELINE_FUNCTIONS_MQTTSEND_PARAMETERS_RETAIN: "true"
WRITABLE_PIPELINE_FUNCTIONS_MQTTSEND_PARAMETERS_PERSISTONERROR: "false"
# WRITABLE_PIPELINE_FUNCTIONS_MQTTSEND_ADDRESSABLE_PUBLISHER:
```

3. Start EdgeX Foundry again
   ```
   docker-compose up -d
   ```

Any messages sent to EdgeX Foundry will now be forwarded to the local MQTT broker,
captured by the messenger app and entered into the InfluxDB database.

Adding Grafana

The final step is to view the data through a Grafana dashboard.

Note: Send some data to EdgeX first to make sure the InfluxDB is populated. It's not possible to
build a dashboard if there's no data in the DB. Use Postman, the Python script or live data from
a DHT sensor for a minute or so to generate the initial data points.

1. Open a browser to: http://<edgex ip>:3000
2. Grafana will ask to create a new admin password
3. After logging in, add an InfluxDB data source. The settings used here correspond to the
   settings used when launching the InfluxDB docker container earlier:
   - URL: http://<edgex ip>:8086
   - Database: sensordata
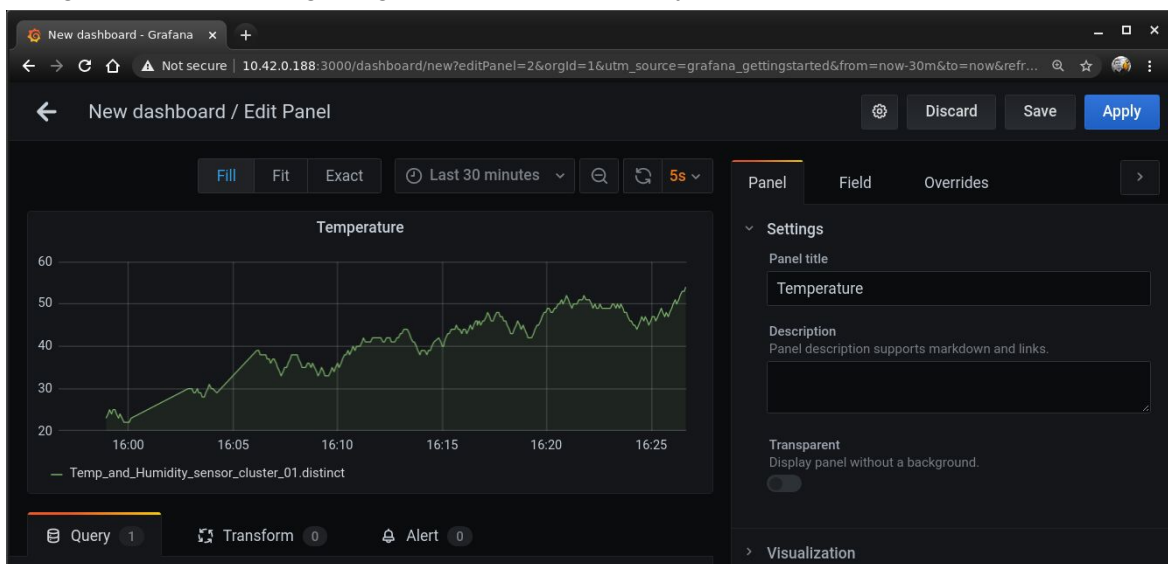   - User: root
   - Password: pass

Test and add the InfluxDB data source.

4. Click "Create your first dashboard" and "Add new panel"
5. Update the statement below the graph as follows:

6. The graph should start getting populated. Click Apply to revert to the main dashboard.

7. Repeat for "humidity". Try a few different types of visualization and ways to display data.

Appendix

Links and references

Main project page: https://www.edgexfoundry.org/
Slack: https://slack.edgexfoundry.org/
Wiki: https://wiki.edgexfoundry.org/
Docs: https://docs.edgexfoundry.org/
Docker Hub: https://hub.docker.com/u/edgexfoundry/
GitHub: https://github.com/edgexfoundry

About the author

Name: Jonas Werner
Currently employed by: Dell Technologies, Tokyo Japan
Jonas's blog: https://jonamiki.com/?s=edgex
Jonas's GitHub: https://github.com/jonas-werner/EdgeX_Tutorial
Comments/feedback: https://jonamiki.com/about/ or "jonas@jonamiki.com" (please be civil -
                        nobody pays me to do this :)
Tags: #edgexfoundry, #iot, #opensource, #grafana, #influxdb, #iwork4dell