Alma Mater Studiorum – University of Bologna
CdS Laurea Magistrale (MSc) in
Computer Science Engineering

Mobile Systems M course (8 ECTS)
II Term – Academic Year 2022/2023

04.lab – Internet of Things (IoT): Hands-on Labs with Azure and EdgeX

Paolo Bellavista
paolo.bellavista@unibo.it

Alma Mater Studiorum – University of Bologna
CdS Laurea Magistrale (MSc) in
Computer Science Engineering

hands on

Azure IoT Hub

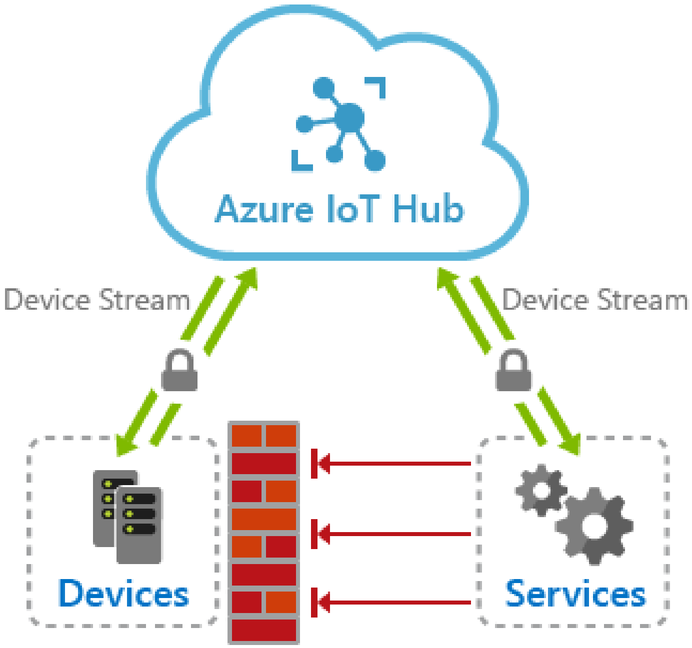
Alberto Cavalucci
alberto.cavalucci2@unibo.it

Agenda

- Recap on Azure IoT technlogies
- Prerequisites and HowTos
- Create an Iot Hub
- Deploy and manage Edgemodule

Azure IoT Hub

IoT Hub is a **cloud-hosted service** that serves as a message hub for bidirectional communication between application and IoT devices



Azure IoT Hub

Azure IoT device SDK libraries are used to build the communication with IoT Hub.

Languages supported:
C
C#
Java
Python

Protocols supported:
HTTPS
AMQP
MQTT

Azure IoT Edge

Service that moves the business logic **from the cloud to the edge** of the architecture. Makes data aggregation and analytics faster being closer to the devices

Three main components:

Edge Modules: containers that run Azure services and apps locally to the device.

Edge Runtime: environment that runs on each device and manages the modules deployed.

Cloud interface: to remotely monitor the devices

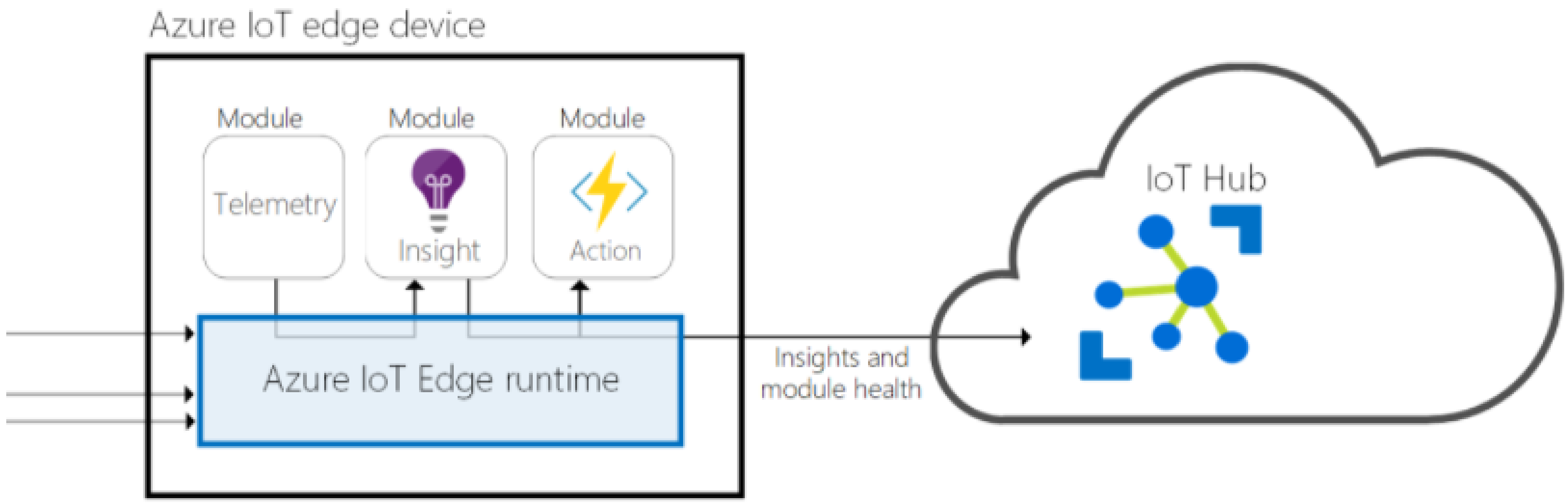
Edge modules

Smallest unit of computation.

Every **module** is made of 4 conceptual elements:

- Image: package containing the software of the module.
- Instance: unit of computation that runs the image on the device. It is started by IoT Runtime.
- Identity: information about credentials and permissions associated with each module.
- Twin: JSON document that stores metadata regarding the status of a module and configuration.

Edge runtime



The **runtime manages deployment and update of the modules**, availability of the services reporting the status to the cloud and communication both with the cloud and the downstream to the devices

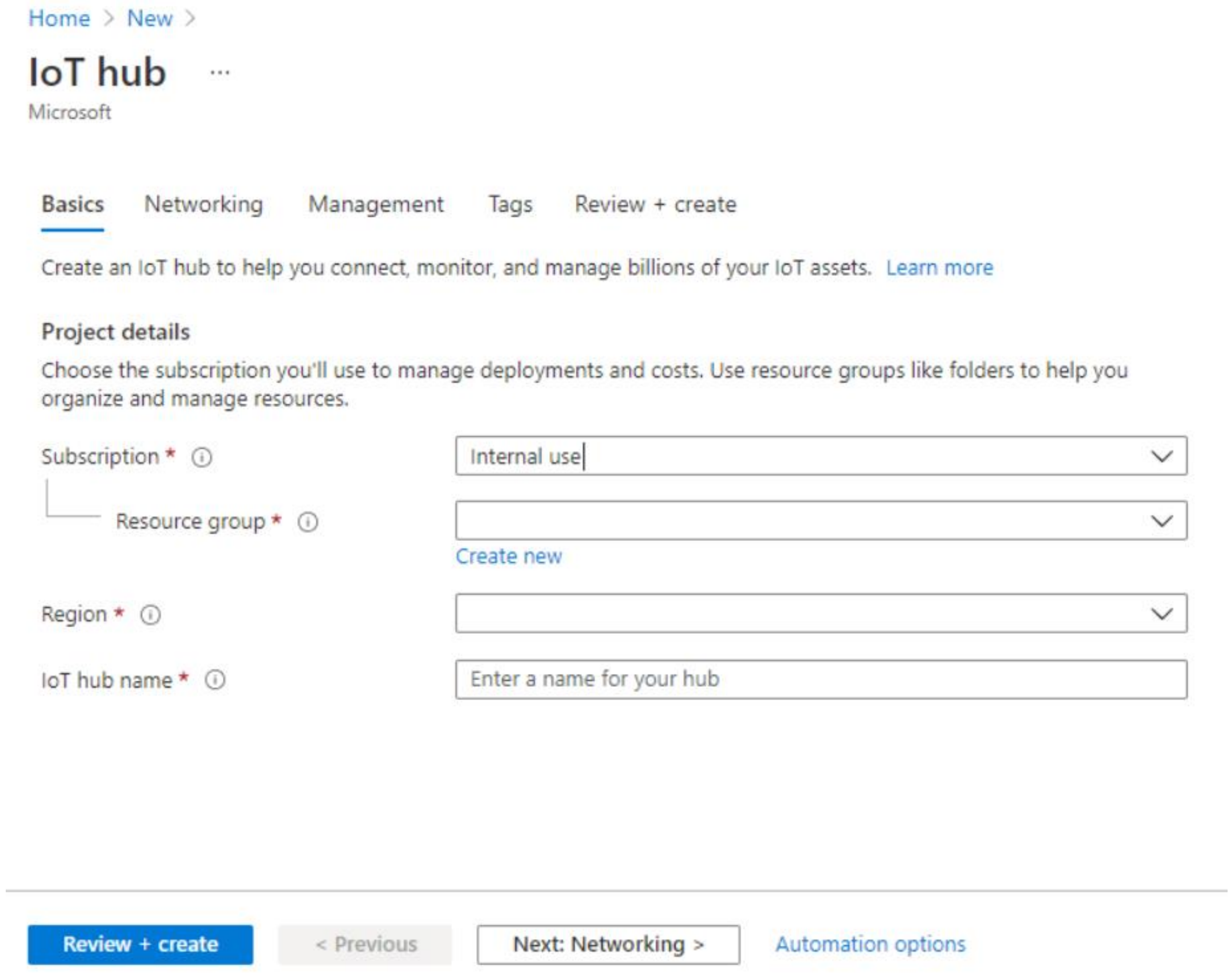
Prerequisites

- Free Azure subscription. https://azure.microsoft.com/en-us/free/?ref=microsoft.com&utm_source=microsoft.com&utm_medium=docs&utm_campaign=visualstudio (no credit card required)
- Install Azure CLI for your platform. https://docs.microsoft.com/en-us/cli/azure/install-azure-cli

Create the IoT Hub (1/3)

1. Sign in to the https://portal.azure.com/
2. Click on the «Create a resource» button and search for Azure IoT Hub under Internet of Things tab
3. Follow the workflow making sure to choose the right options on the basic tab

Create the IoT Hub (2/3)



IoT hub ···

Microsoft

Create an IoT hub to help you connect, monitor, and manage billions of your IoT assets. Learn more

Project details

Choose the subscription you'll use to manage deployments and costs. Use resource groups like folders to help you organize and manage resources.

Subscription * ⓘ: Internal use

Resource group * ⓘ:

Create new

Region * ⓘ:

IoT hub name * ⓘ: Enter a name for your hub

Create the IoT Hub (3/3)

Subscription: choose the Free tier one.

Resource group: choose the option to create a new one and select a name. This is gonna be used for all the resources allocated in this lab.

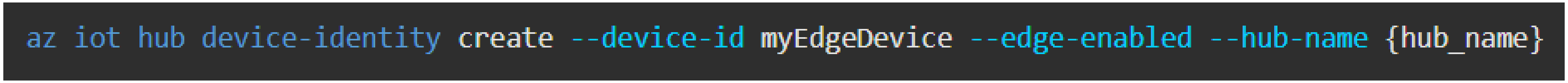
IoT Hub name: unique name for the hub utilized to create the connection.

Region: region where is located the hub.

Register the IoT Edge device (1/2)

We want to create a device identity, which is a «virtual» version of the edge device. It has the same properties of the real device and is connected to it trough a connection string.

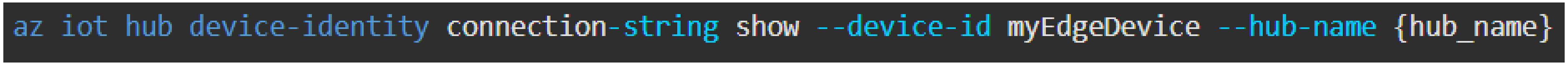
1. In the Azure CLI we enter the following command to create an EdgeDevice.



```
az iot hub device-identity create --device-id myEdgeDevice --edge-enabled --hub-name {hub_name}
```

Register the IoT Edge device (2/2)

2. With the creation of the device also the connection string and the shared key have been created. Insert the next command to see the connection string that will be required later in the lab



```
az iot hub device-identity connection-string show --device-id myEdgeDevice --hub-name {hub_name}
```

```
{
        "connectionString": "HostName ={hub_name}.azure-devices.net;
        DeviceId=myEdgeDevice;
        SharedAccessKey={Key}"
}
```

Install Azure IoT Edge on device(1/3)

Edge runtime is what makes a device an IoT edge device. It can be installed in different types of machines, in this case we are going to use a Raspberry Pi.

First we are going to set and download microsoft package configuration

```
curl https://packages.microsoft.com/config/debian/stretch/multiarch/prod.list
> ./microsoft-prod.list
sudo cp ./microsoft-prod.list /etc/apt/sources.list.d/
curl https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor >
microsoft.gpg
sudo cp ./microsoft.gpg /etc/apt/trusted.gpg.d/
```

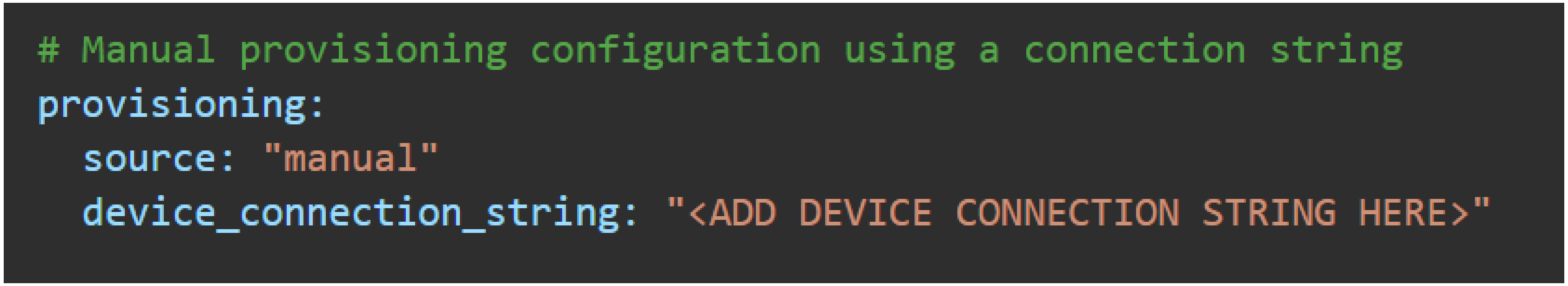
Install Azure IoT Edge on device(2/3)

Now we are going to **install the container engine that will host IoT edge services and the runtime**.
Moby engine is the only supported container engine for IoT edge, altough is based on Docker and is compatible with Docker Image

```
sudo apt-get update
sudo apt-get install moby-engine
sudo apt-get install aziot-edge
```

Install Azure IoT Edge on device(3/3)

Once all the software needed is installed the connection string we produced earlier has to be set in the /etc/aziot/config.toml config file.

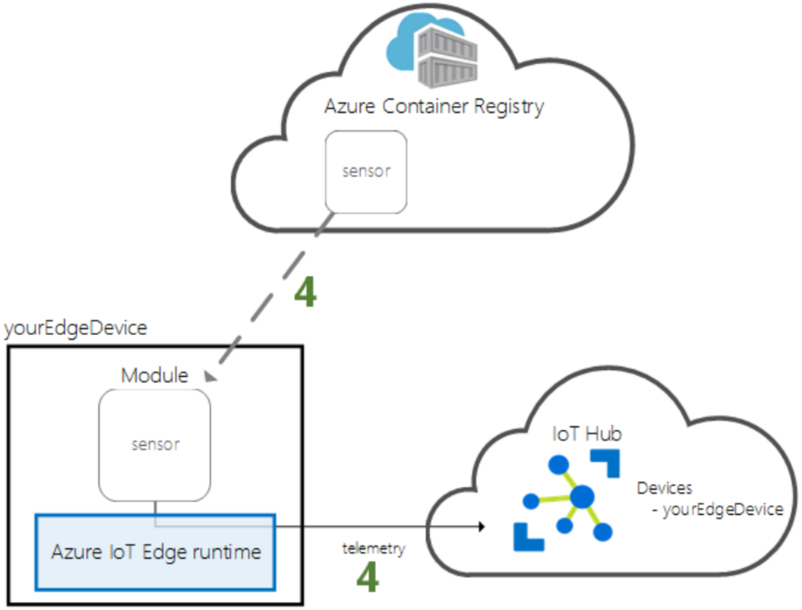


```
# Manual provisioning configuration using a connection string
provisioning:
  source: "manual"
  device_connection_string: "<ADD DEVICE CONNECTION STRING HERE>"
```

After a restart our edge device is ready to use.

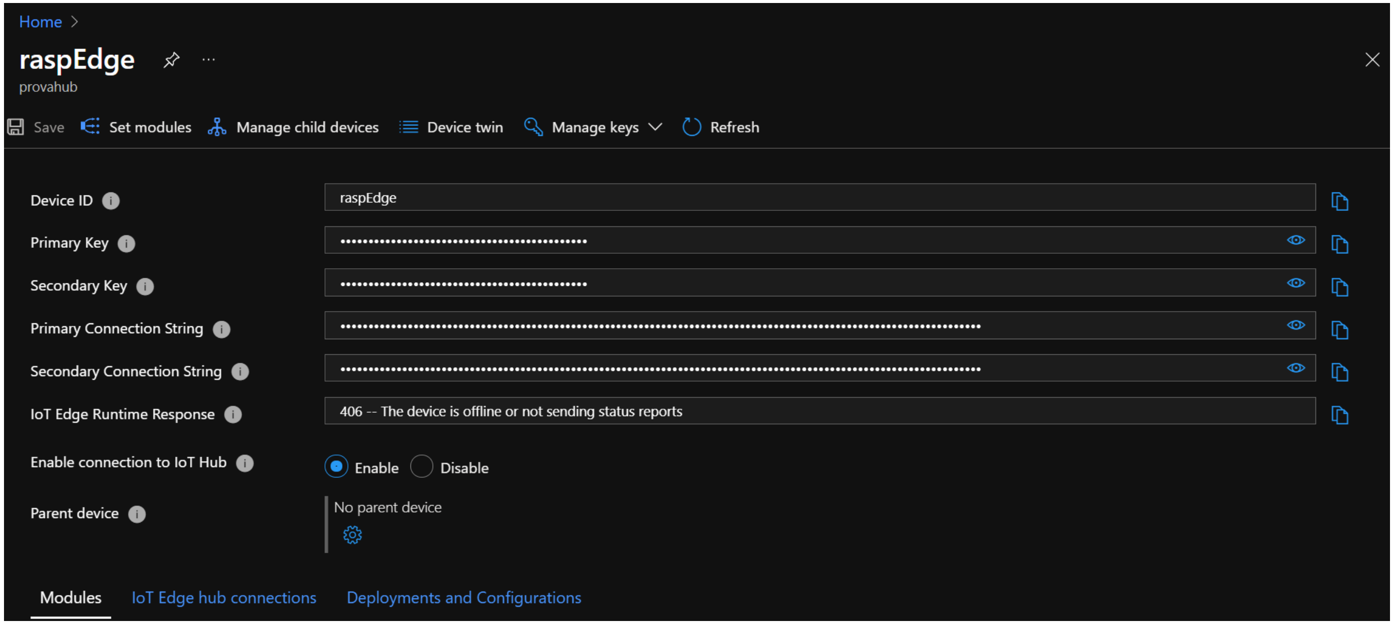
```
sudo iotedge config apply
```

Deployment of the module



After the creation of the hub and the installation of the runtime, we want to **deploy remotely the module** from Azure portal to the IoT device

Deployment of the module



Device ID: raspEdge

Primary Key: ••

Secondary Key: ••

Primary Connection String: ••

Secondary Connection String: ••

IoT Edge Runtime Response: 406 -- The device is offline or not sending status reports

Enable connection to IoT Hub: ● Enable ○ Disable

Parent device: No parent device

1. Log in to Azure portal, go to your IoT hub -> Automatic Device Management -> IoT Edge
2. Select the device ID and then “set modules”

Deployment of the module

At this point, we will follow the workflow to deploy a module, for this demo we used a simulated temperature sensor already present in the marketplace. To use it we click «ADD» and then marketplace module. When we finish the flow under the tab «modules» we should see two more modules in addition to the edgeAgent

Name	Type	Specified in Deployment	Reported by Device	Runtime Status	Exit C...
$edgeAgent	IoT Edge System Module	✓ Yes	✓ Yes	unknown	0
$edgeHub	IoT Edge System Module	✓ Yes	✓ Yes	unknown	0
SimulatedTemperatureSensor	IoT Edge Custom Module	✓ Yes	✓ Yes	unknown	0

Read the data

The simulated sensor is up and running. Now, we want to write a script that connects to the Hub and reads the data simulating the cloud layer. First we need to note this connection parameters

```
az iot hub show --query properties.eventHubEndpoints.events.endpoint
--name {YourIoTHubName}

az iot hub show --query properties.eventHubEndpoints.events.path --
name {YourIoTHubName}

az iot hub policy show --name service --query primaryKey --hub-name
{YourIoTHubName}
```

Read the data

```
CONNECTION_STR =
f'Endpoint={EVENTHUB_COMPATIBLE_ENDPOINT}/;\
SharedAccessKeyName=service;\
SharedAccessKey={IOTHUB_SAS_KEY};\
EntityPath={EVENTHUB_COMPATIBLE_PATH}'

client = EventHubConsumerClient.from_connection_string(
        conn_str=CONNECTION_STR,
        consumer_group="$default"
              )
```

The parameters will form the connection string with which we create a consumer for the hub.

Read the data

```
try:
    with client:
        client.receive_batch(
            on_event_batch=on_event_batch,
            on_error=on_error)
except KeyboardInterrupt:
    print("Receiving has stopped.")
```

In order to consume the events from the hub we have to invoke the method «receive_batch». The arguments are two callback functions that will be executed depending on the success or the failure of the invocation

Read the data

```
def on_event_batch(partition_context, events):
    for event in events:
        print("Telemetry received: ", event.body_as_str())
    partition_context.update_checkpoint()
```

The callback firstly consumes all the event received from the hub, in this case just printing the body of the message, and then updates with a checkpoint for the next call of the method

Alma Mater Studiorum – University of Bologna
CdS Laurea Magistrale (MSc) in
Computer Science Engineering

hands on
EDGEXFOUNDRY™

Gianluca Rosi
gianluca.rosi3@unibo.it

What’s EdgeX Foundry

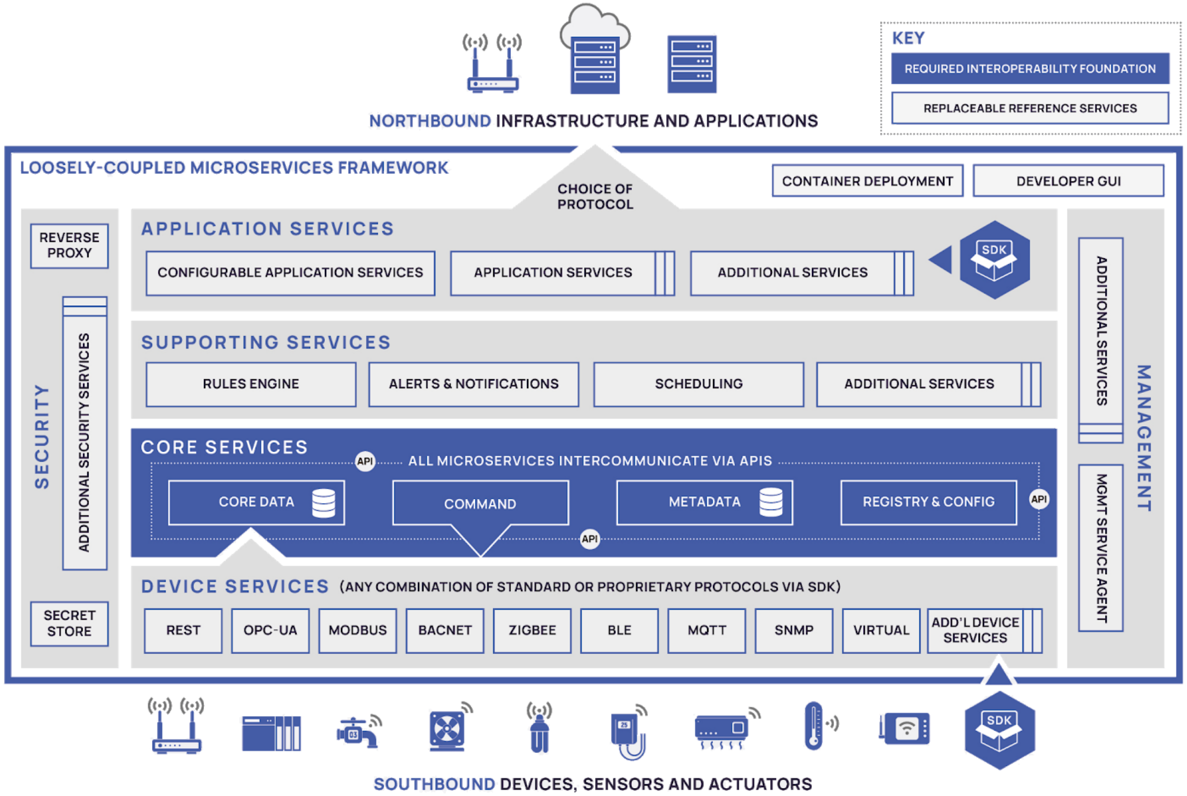
- It’s a multi-platform, open source software (written in Golang), dedicated tomake more unfirm **Industrial IIoT** communication protocols
- Freshly developed from **Dell** code-base for their own edge gateways and hosted by the **Linux Foundation** as a project on LF Edge

LF
EDGE

EDGEXFOUNDRY™

How does it work?

- The **Core Services** are coordinating every event and their reaction, based on the stored knowledge



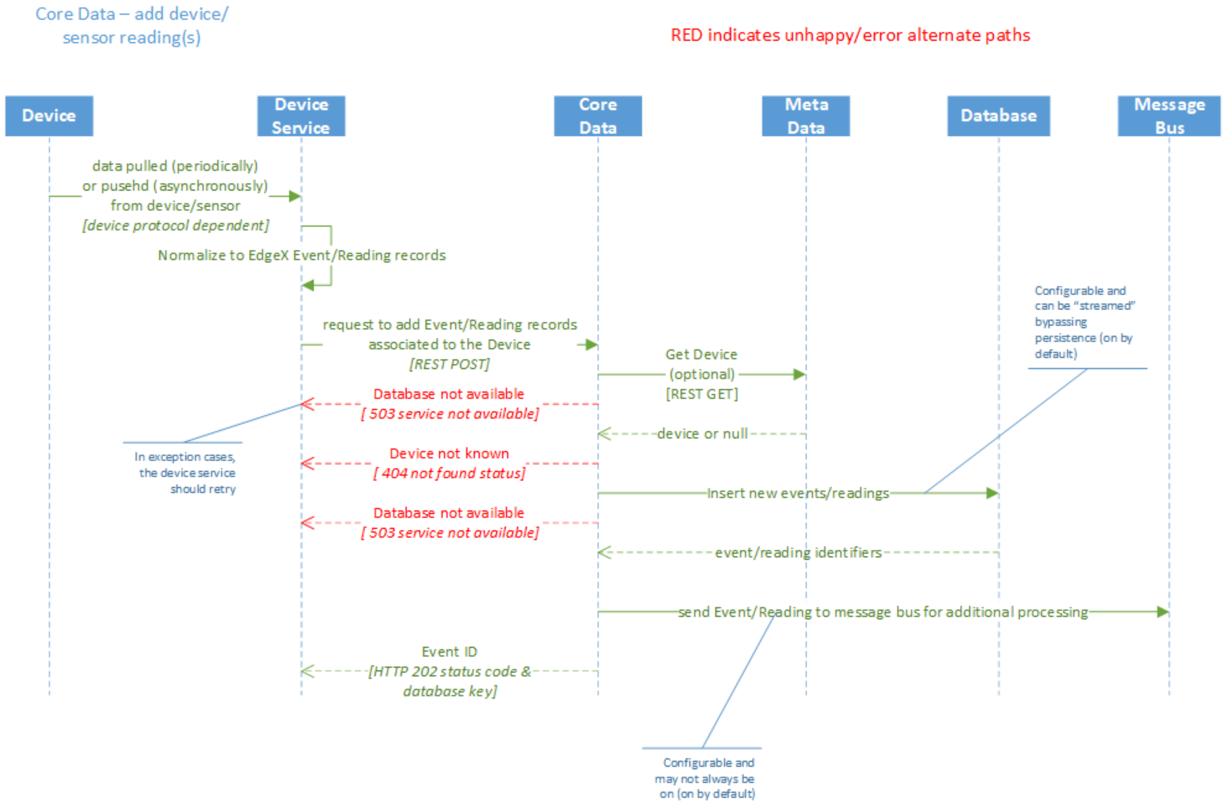
- Messages are flowing from bottom to top and vice versa, making these 4 microservices act as interface among the **north-side** and **south-side**

Core Data

It stores all data sent through EdgeX framework (may be disabled for stream-only) with Redis

Once received, events are then published via ZeroMQ to Application Services

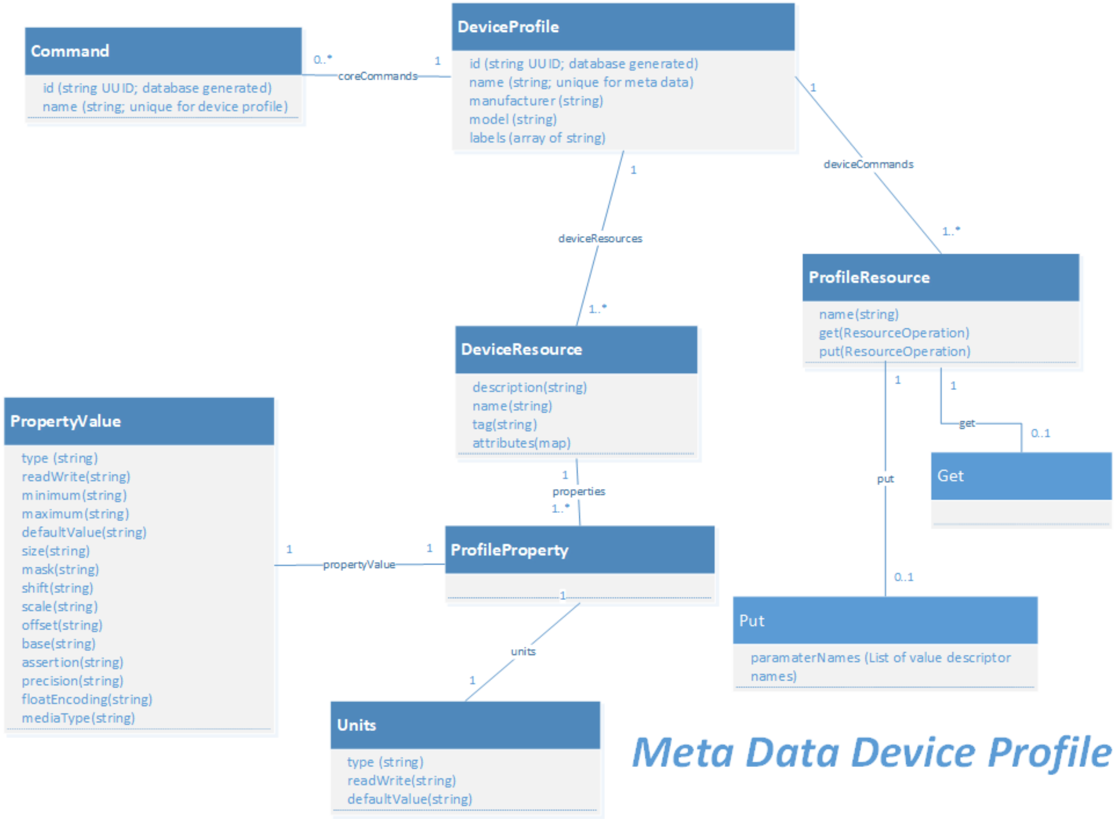
A scheduled work is in charge to clean correctly exported data, thus to free memory for new messages



Core Metadata

Stores the knowledge of every registered device and sensors, this lets the framework to know which resources are available

Still based on Redis, device profiles have to be provided in YAML files

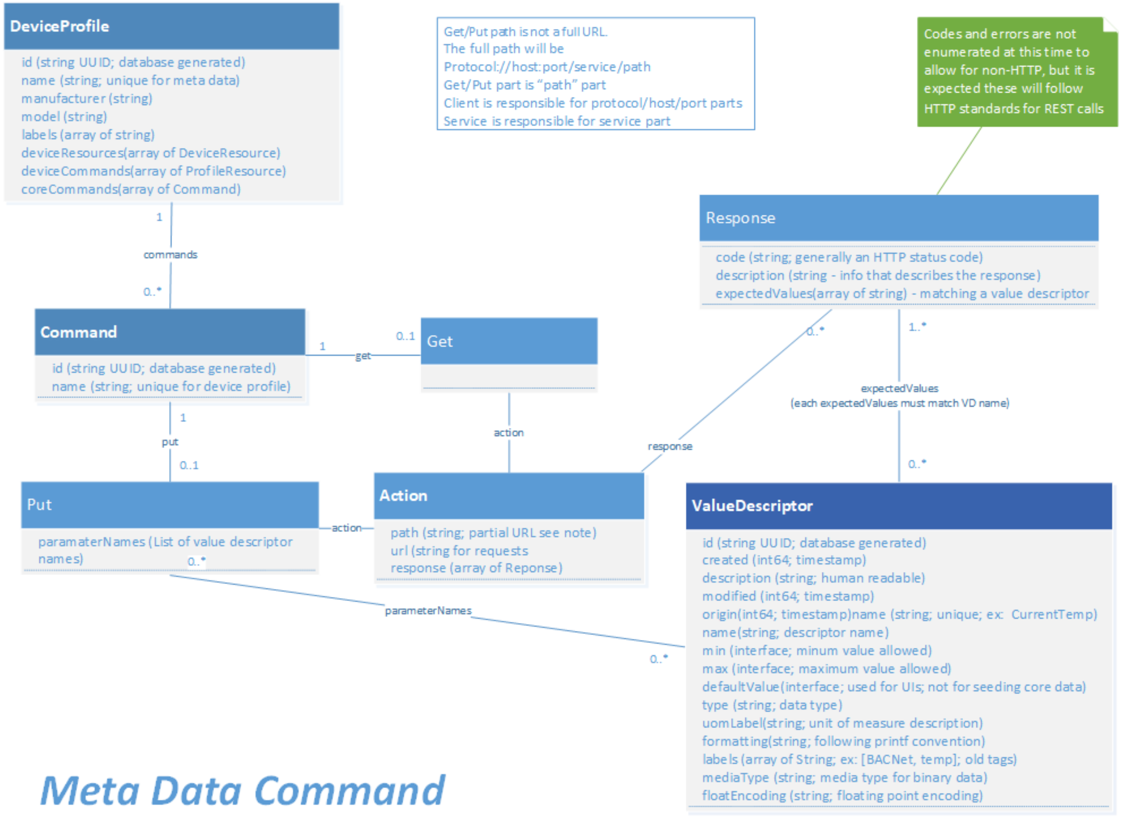


Meta Data Device Profile

Core Command

This microservice is a proxy service for action requests from the (north) exposed REST API to the Device Services, which are the only in charge to directly talk to devices

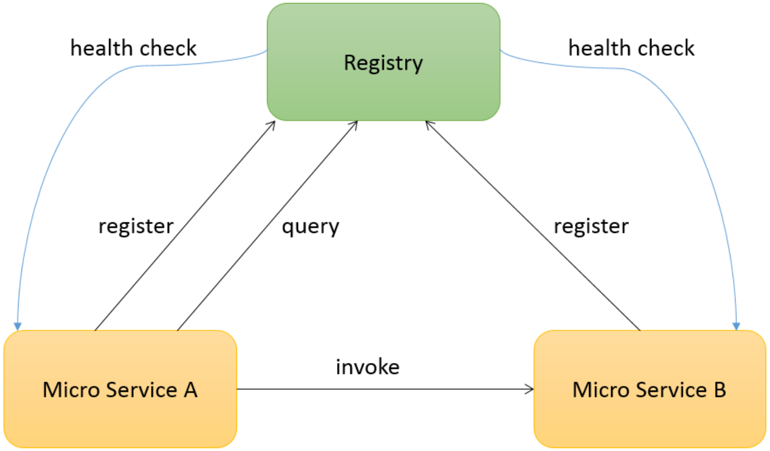
Metadata microservice provides all Core Command knowledge



Meta Data Command

Registry and Config

The EdgeX registry and configuration service provides other EdgeX Foundry micro services with information about ***associated services within EdgeX Foundry (such as location and status) and configuration properties*** (i.e. - a repository of initialization and operating values)



Registry:
microservices status and health monitor (Consul)

Config:
usually provided in TOML file, useful for static parameters on microservices

let’s get practical 0. **up and running**

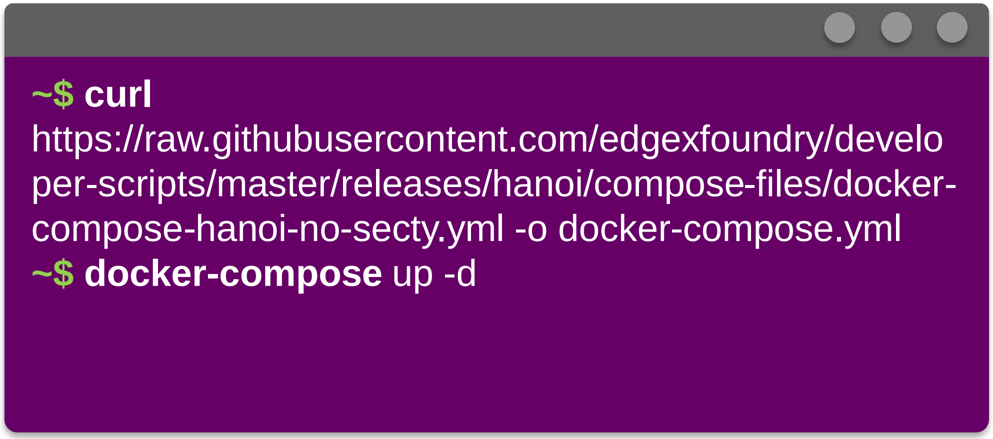
docs @ https://docs.edgexfoundry.org/1.3/getting-started/quick-start/

Installing EdgeX Foundry with Docker

This tutorial is based on a Linux env. (Kubuntu 20.04)
Suggestion: it’s ok to use a VM with as little as 2 cores and 2GB of RAM

The fastest way to start running EdgeX is by using pre-built Docker images. To use them you'll need to install the following:

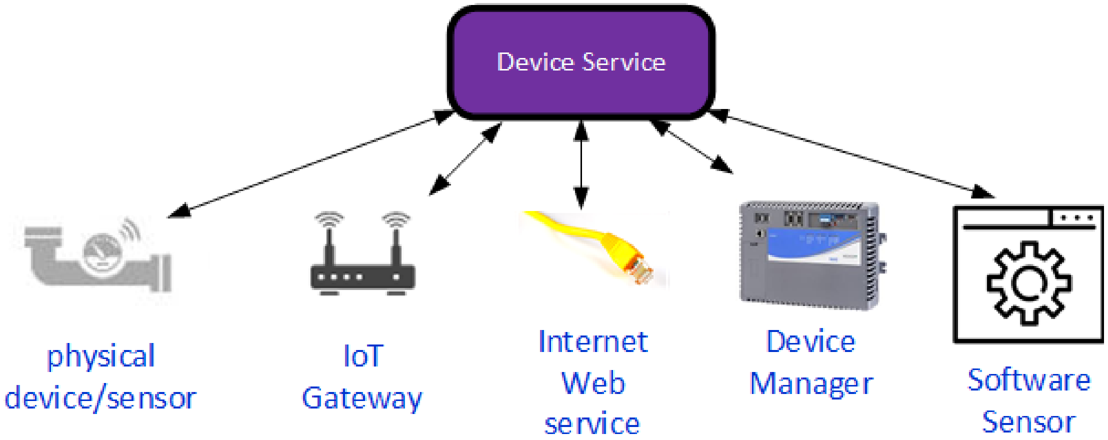
1. Docker engine docker.com/get-started
2. Docker Compose docs.docker.com/compose/install
3. download / save the latest docker-compose file and rename
4. issue command to download and run the EdgeX Foundry Docker images from Docker Hub:



```
~$ curl
https://raw.githubusercontent.com/edgexfoundry/develo
per-scripts/master/releases/hanoi/compose-files/docker-
compose-hanoi-no-secty.yml -o docker-compose.yml
~$ docker-compose up -d
```

let’s get practical 1. **managing data**

From *data sources*
to *events*



software abstraction of physical devices:

pre-defined or custom Device Services (SDK provided) allows to translate device communication protocols to a standard message (event) readable by EdgeX Foundry framework

- This allows to register the source device and automatically add **metadata** on its messages and operations

docs @ https://docs.edgexfoundry.org/1.3/microservices/device/Ch-DeviceServices/

Connect a device with Modbus and MQTT (1)

Add in the descriptor file the device services (already in the EdgeX framework), with only a 2-space indent on the first line (be careful to maintain the proposed indentation)

docker-compose.yml

```
device-modbus:
 image: edgexfoundry/docker-device-modbus-go:1.2.1
 ports:
  - 127.0.0.1:49991:49991/tcp
 container_name: edgex-device-modbus
 hostname: edgex-device-modbus
 networks:
  edgex-network: {}
 environment:
  [chk the provided file]
 depends_on:
 - data
 - command
```

```
device-mqtt:
 image: edgexfoundry/docker-device-mqtt-go:1.2.1
 ports:
  - 127.0.0.1:49982:49982/tcp
 container_name: edgex-device-mqtt
 hostname: edgex-device-mqtt
 networks:
  edgex-network: {}
 environment:
  [chk the provided file]
 depends_on:
   - data
   - command
```

Connect a device with Modbus and MQTT (2)

Enable from the descriptor file the device services (already in the EdgeX framework)

```
~$ docker-compose up -d device-mqtt
```

```
~$ docker-compose up -d device-modbus
```

let's get practical 2. **working on a use-case**

Register a device by a Device profile (Modbus)

HMIsim.yml

It’s a YAML file that can be sent in anytime when EdgeX is up and running. The upload coincides with the device registration.

- This file contains **metadata** and **operations** supported by the device

```
name: "HMI_6k"
manufacturer: "SACMI"
model: "XYZ145"
description: "Dispositivo HMIsimulator"
labels:
  - "modbus"
  - "interface"
  - "simulator"
deviceResources:
    -
        name: "DatiCiclo"
        description: "Dati ciclo"
        attributes:
            { primaryTable:
"HOLDING_REGISTERS", startingAddress: "2"}
        properties:
          value:
            { type: "UINT16", scale: "1"}
          units:
            { type: "String", readWrite: "R",
              defaultValue: "min"}
```

[cont...]

docs @ https://docs.edgexfoundry.org/1.3/microservices/device/profile/Ch-DeviceProfileRef/

HMIsim.yml

```
[...cont]
deviceCommands:
  -
    name: "DatiCiclo"
    get:
      - { index: "1", operation: "get", deviceResource: "DatiCiclo" }
coreCommands:
  -
    name: "DatiCiclo"
    get:
      path: "/api/v1/device/{deviceId}/DatiCiclo"
      responses:
        -
          code: "200"
          description: "Get the DatiCiclo"
          expectedValues: ["DatiCiclo"]
        -
          code: "500"
          description: "internal server error"
          expectedValues: []
```

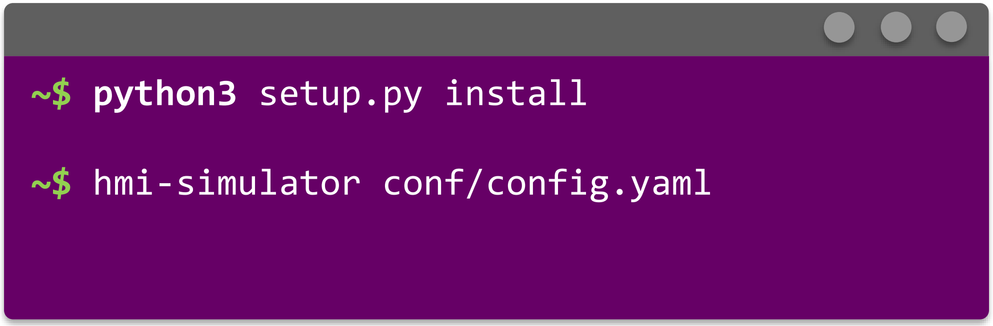
Now we have to make EdgeX be aware of a “future” device with the described features by uploading that description file (CLI command provided in CLI_JSON_desc.txt)

```
~$ curl http://localhost:48081/api/v1/deviceprofile/uploadfile \
  -F "file=@HMIsim.yaml"
```

Run the devices (Modbus)

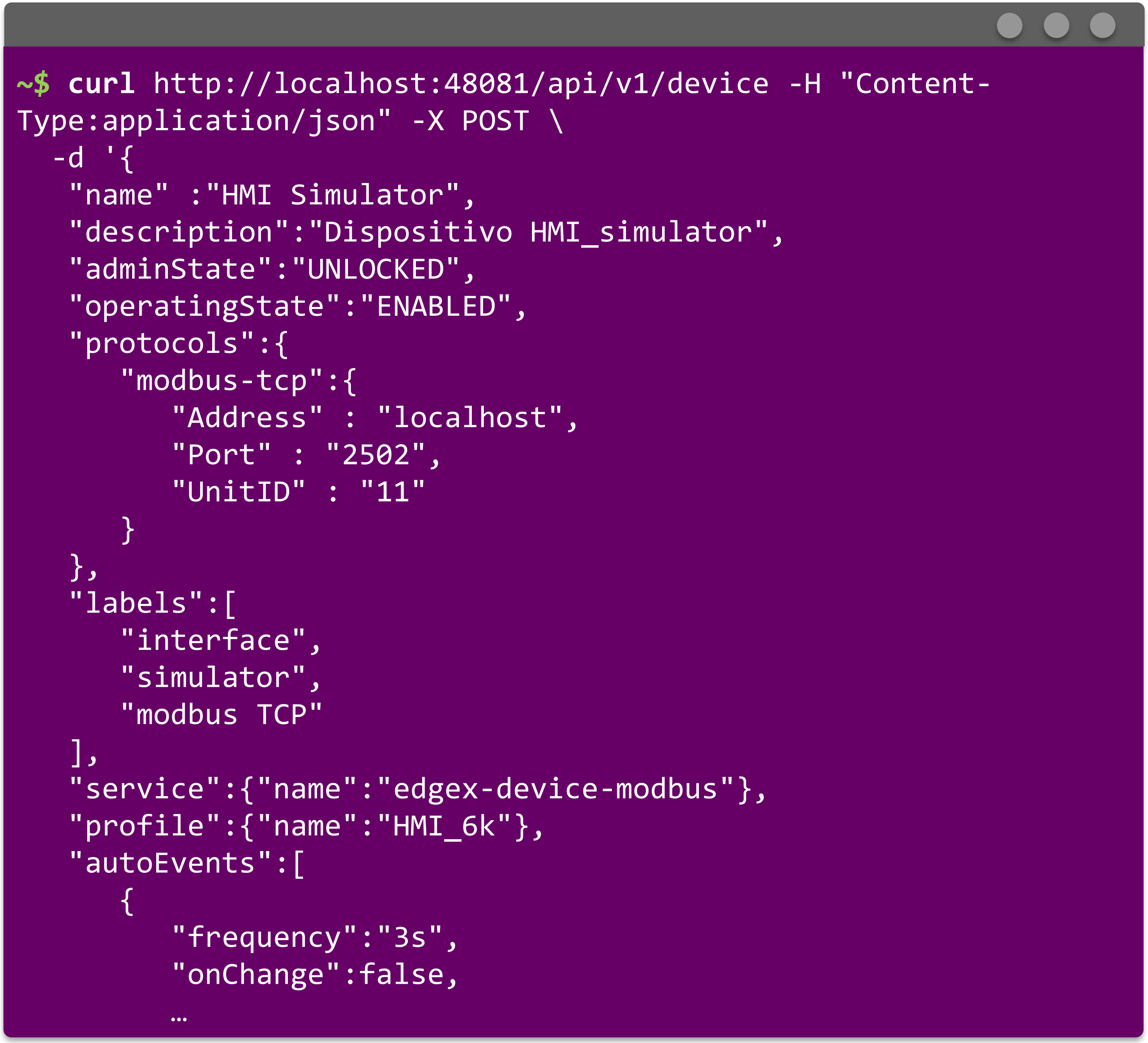
This is a simulator of a real production machine

Unpack the zip file, install and run with the following commands:



```
~$ python3 setup.py install

~$ hmi-simulator conf/config.yaml
```



```
~$ curl http://localhost:48081/api/v1/device -H "Content-
Type:application/json" -X POST \
  -d '{
   "name" :"HMI Simulator",
   "description":"Dispositivo HMI_simulator",
   "adminState":"UNLOCKED",
   "operatingState":"ENABLED",
   "protocols":{
      "modbus-tcp":{
         "Address" : "localhost",
         "Port" : "2502",
         "UnitID" : "11"
      }
   },
   "labels":[
      "interface",
      "simulator",
      "modbus TCP"
   ],
   "service":{"name":"edgex-device-modbus"},
   "profile":{"name":"HMI_6k"},
   "autoEvents":[
      {
         "frequency":"3s",
         "onChange":false,
         …
```

Then we can register a new physical device

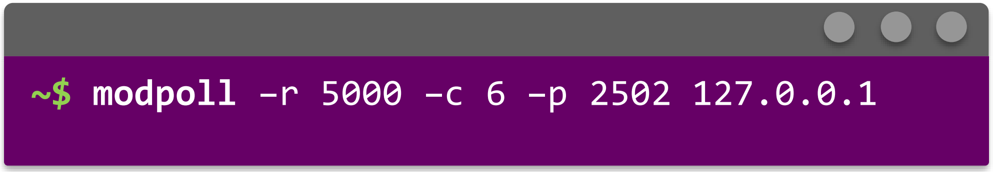
CLI command is in

CLI_json_desc.txt

let’s get practical 3. **exporting data**

Read the devices (Modbus) [1]

Let’s check that our generator is working:



```
~$ modpoll -r 5000 -c 6 -p 2502 127.0.0.1
```

modbusdriver.com/modpoll.html

Read the devices (Modbus) [2]

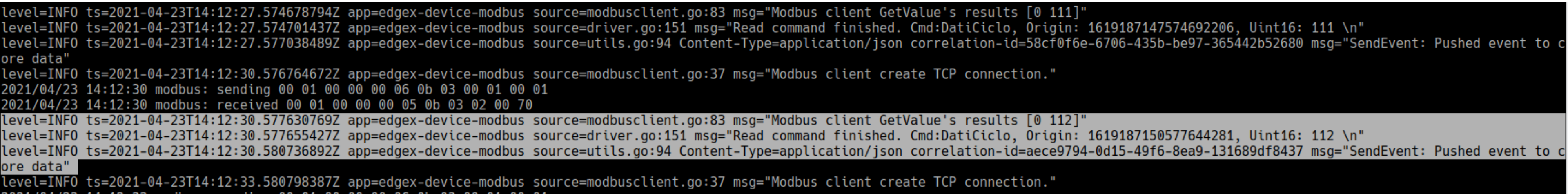
Now we can check data flowing through EdgeX by checking the **device-modbus** log and querying the **core-data** to get some readings with an HTTP GET call to*:

http://localhost:48080/api/v1/event/{start}/{end}/{limit}

level=INFO ts=2021-04-23T14:12:30.577630769Z app=edgex-device-modbus source=modbusclient.go:83 msg="Modbus client GetValue's results [0 112]"

level=INFO ts=2021-04-23T14:12:30.577655427Z app=edgex-device-modbus source=driver.go:151 msg="Read command finished. Cmd:**DatiCiclo**, Origin: 1619187150577644281, **Uint16: 112** \n"

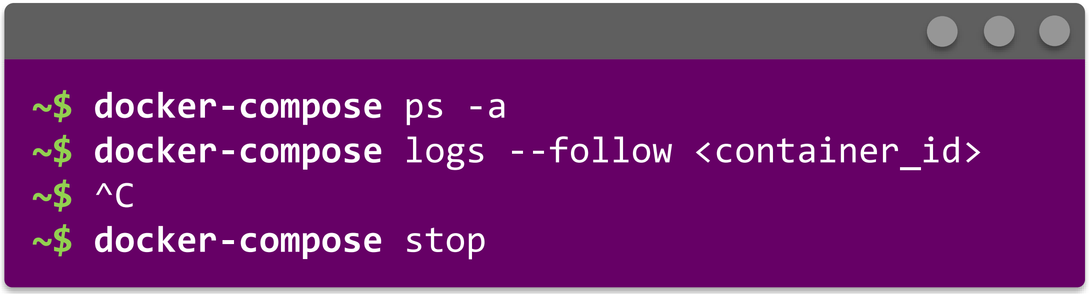
level=INFO ts=2021-04-23T14:12:30.580736892Z app=edgex-device-modbus source=utils.go:94 Content-Type=application/json correlation-id=aece9794-0d15-49f6-8ea9-131689df8437 msg="**SendEvent: Pushed event to core data**"



```
level=INFO ts=2021-04-23T14:12:27.574678794Z app=edgex-device-modbus source=modbusclient.go:83 msg="Modbus client GetValue's results [0 111]"
level=INFO ts=2021-04-23T14:12:27.574701437Z app=edgex-device-modbus source=driver.go:151 msg="Read command finished. Cmd:DatiCiclo, Origin: 1619187147574692206, Uint16: 111 \n"
level=INFO ts=2021-04-23T14:12:27.577038489Z app=edgex-device-modbus source=utils.go:94 Content-Type=application/json correlation-id=58cf0f6e-6706-435b-be97-365442b52680 msg="SendEvent: Pushed event to c
ore data"
level=INFO ts=2021-04-23T14:12:30.576764672Z app=edgex-device-modbus source=modbusclient.go:37 msg="Modbus client create TCP connection."
2021/04/23 14:12:30 modbus: sending 00 01 00 00 00 06 0b 03 00 01 00 01
2021/04/23 14:12:30 modbus: received 00 01 00 00 00 05 0b 03 02 00 70
level=INFO ts=2021-04-23T14:12:30.577630769Z app=edgex-device-modbus source=modbusclient.go:83 msg="Modbus client GetValue's results [0 112]"
level=INFO ts=2021-04-23T14:12:30.577655427Z app=edgex-device-modbus source=driver.go:151 msg="Read command finished. Cmd:DatiCiclo, Origin: 1619187150577644281, Uint16: 112 \n"
level=INFO ts=2021-04-23T14:12:30.580736892Z app=edgex-device-modbus source=utils.go:94 Content-Type=application/json correlation-id=aece9794-0d15-49f6-8ea9-131689df8437 msg="SendEvent: Pushed event to c
ore data"
level=INFO ts=2021-04-23T14:12:33.580798387Z app=edgex-device-modbus source=modbusclient.go:37 msg="Modbus client create TCP connection."
```

*reference @ https://app.swaggerhub.com/apis/EdgeXFoundry1/core-data/1.2.1#/default/

Clean shutdown & utilities

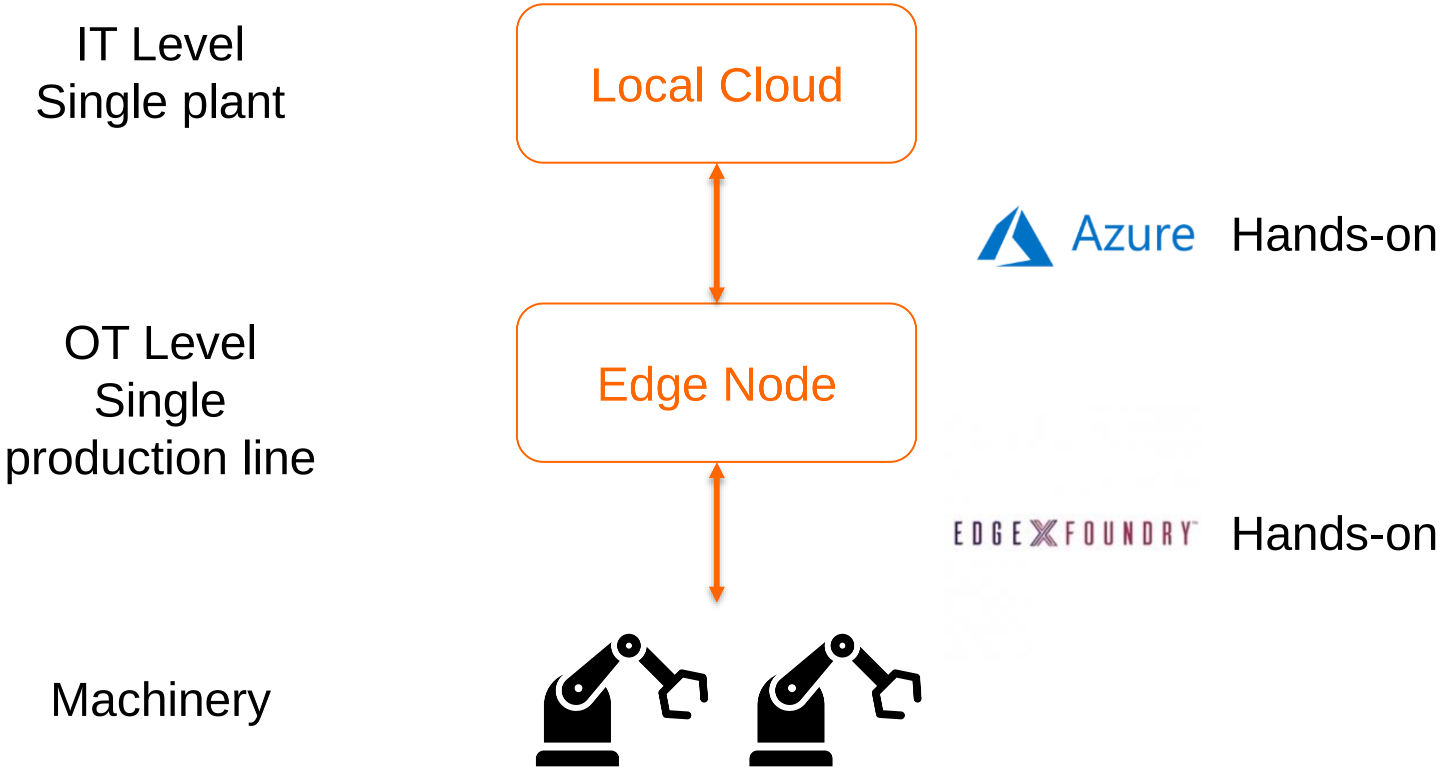


```
~$ docker-compose ps -a
~$ docker-compose logs --follow <container_id>
~$ ^C
~$ docker-compose stop
```

- Consul UI si at **localhost:8500/ui**
- Every HTTP request can be executed by your favourite HTTP API client app (like Postman or Insomnia)
- API reference is at https://app.swaggerhub.com/search?type=API&owner=EdgeXFoundry1
- Hands on based on the formal tutorial at https://docs.edgexfoundry.org/1.3/examples/LinuxTutorial/EdgeX-Foundry-tutorial-ver1.1.pdf

Port	Service
48080	Core Data
48081	Core Metadata
48082	Core Command

To sum up



IT Level
Single plant

Local Cloud

Azure Hands-on

OT Level
Single
production line

Edge Node

EDGEXFOUNDRY Hands-on

Machinery