
fogflow tutorial

Release v3.2.8

Bin Cheng

Nov 29, 2022

INTRODUCTION

1	Motivation	3
2	High level view	5
3	Technical benefit	7
4	Differentiation	9
5	Quick Start	11
6	Hello World Example	15
7	Core concepts	21
8	Programming Model	27
9	API Walkthrough	61
10	System Setup	93
11	Monitoring	101
12	Security	107
13	Compile the source code	125
14	Test	127
15	Related publications	129
16	Troubleshooting	131
17	Contact	133

FogFlow is an IoT edge computing framework that automatically orchestrates dynamic data processing flows over cloud and edges based on various **context**, including:

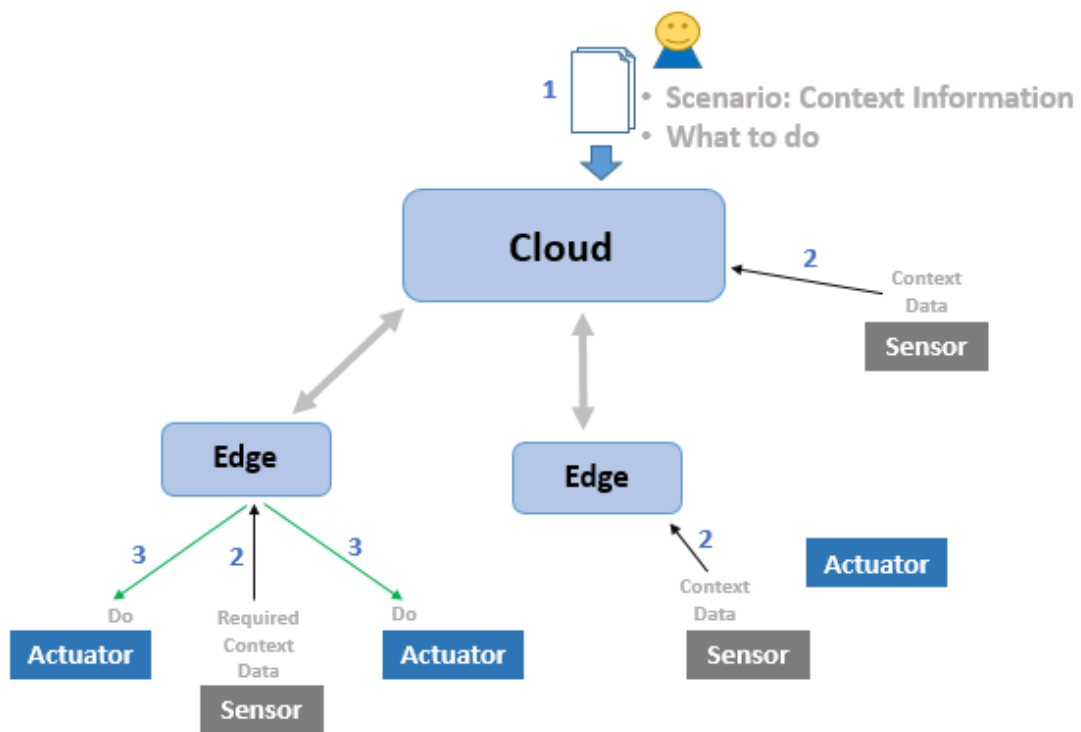
- *system context*: the available system resources from all layers;
- *data context*: the registered metadata of all available data entities;
- *usage context*: the expected usage intention defined by users in terms QoS, latency, and bandwidth cost;

Thanks to its advanced intent-based programming model and context-driven service orchestration, FogFlow is able to **provide optimized QoS with minimal development effort and nearly zero operation overhead**. Currently, FogFlow has been applied into various business use cases in the areas of retails, smart cities, and smart industry.

Nowadays IoT infrastructure providers for smart city, smart industry, and connected vehicles are facing lots of complexity and high operation cost to manage their geo-distributed infrastructures for supporting various IoT services, especially those that require low latency. FogFlow is a distributed execution framework to dynamically orchestrate IoT services over cloud and edges, in order to reduce internal bandwidth consumption and offer low latency and fast response time.

By providing automated and optimized IoT service orchestration with high scalability and reliability, FogFlow helps infrastructure providers to largely reduce their operation cost. FogFlow also provides an intent-based programming model and the development tools for service developers and system integrators to quickly realize IoT services with low development cost and fast time-to-market.

The overall architecture view of FogFlow system is illustrated in below Figure. Infrastructure resources are vertically divided as cloud, edge nodes, sensor devices and Actuators. Computationally intensive tasks, such as data analytics can be performed on the cloud servers, while some tasks, such as stream processing can be effectively moved to the edge nodes (e.g., IoT gateways or endpoint devices with computation capabilities). Sensor shares observed data (observed by different sensor devices) with broker present on edge nodes and edge nodes compute this data based on logic, then sends response to Actuators.



MOTIVATION

FogFlow is an IoT edge computing framework, which is designed to address the following concerns:

- The cost of a cloud-only solution is too high to run a large scale IoT system with >1000 geo-distributed devices
- Many IoT services require fast response time, such as <10ms end-to-end latency
- Service providers are facing huge complexity and cost to fast design and deploy their IoT services in a cloud-edge environment - business demands are changing fast over time and service providers need to try out and release any new services over their shared cloud-edge infrastructure at a fast speed
- Lack of programming model to fast design and deploy IoT services over geo-distributed ICT infrastructure
- Lack of interoperability and openness to share and reuse data and derived results across various applications

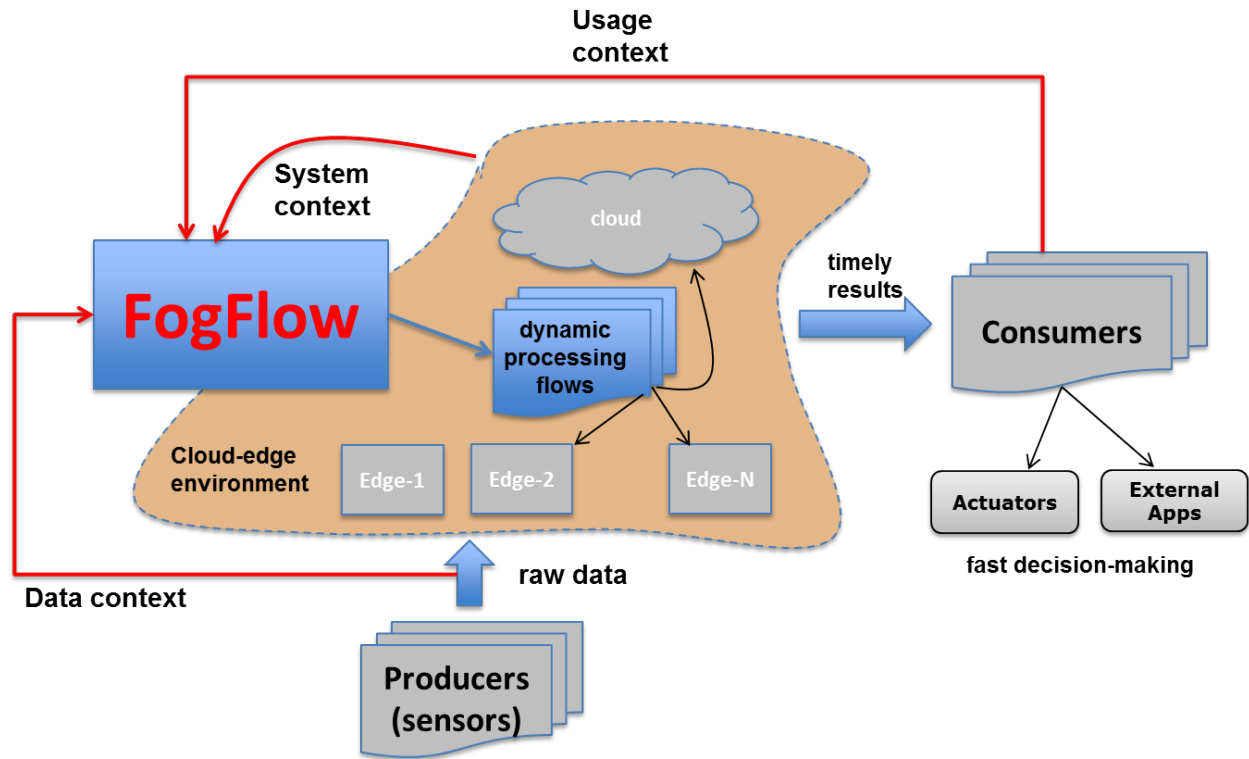
With FogFlow, service providers can easily programme their IoT services with minimal development effort and also they are able to release new IoT services with fast time-to-market. On the other hand, for IoT platform operators, FogFlow can help them to seamlessly server latency-sensitive IoT services over cloud and edges with low operation cost and optimized QoS.

HIGH LEVEL VIEW

The unique feature of FogFlow is **context-driven**, meaning that FogFlow is able to orchestrate dynamic data processing flows over cloud & edges based on three types of contexts, including:

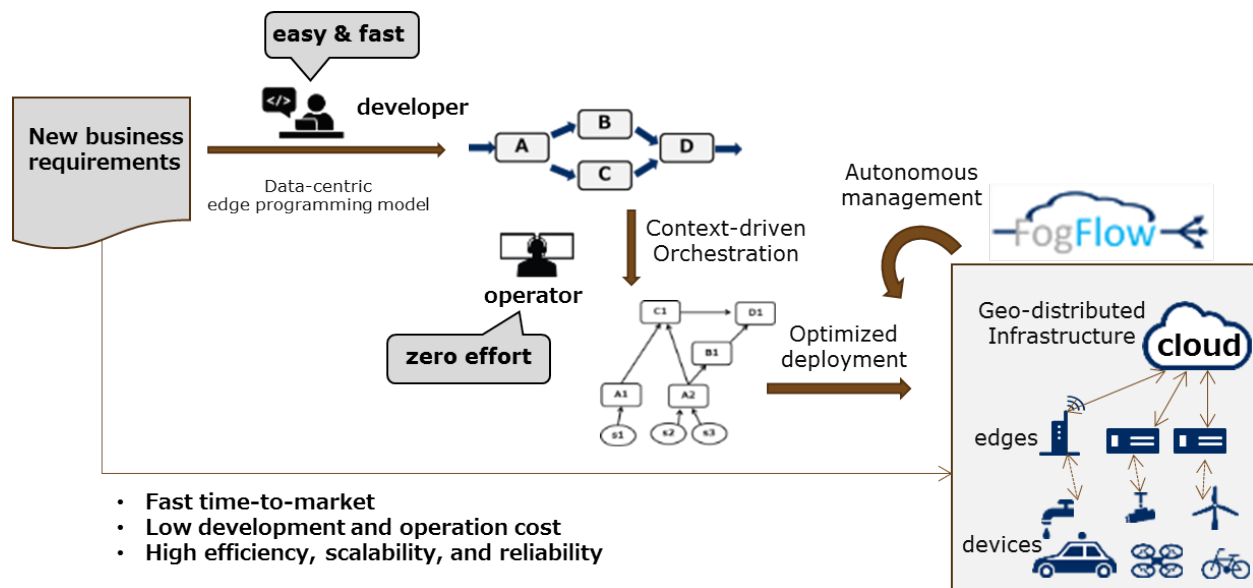
- **System context: available resources which are changing over time**
The resources in a cloud-edge environment are geo-distributed in nature and they are dynamically changing over time; As compared to cloud computing, resources in such a cloud-edge environment are more *heterogenous* and *dynamical*.
- **Data context: the structure and registered metadata of available data, including both raw sensor data and intermediate data**
based on the standardized and unified data model and communicatino interface, namely **NGSI**, FogFlow is able to see the content of all data generated by sensors and data processing tasks in the system, such as data type, attributes, registered metadata, relations, and geo-locations;
- **Usage context: high level intents defined by all different types of users (developers, service consumers, data providers) to specify what they want to achieve.**
For example, for service consumers, they can specify which type of results are expected under which type of QoS within which geo-scope; for data providers, they can specify how their data should be utilized by whom. In FogFlow, orchestration decisions are made to meet those user-definable objectives during the runtime. We are working on more advanced algorithms to enable those new features.

By leveraging these three kinds of context, FogFlow is able to orchestrate IoT services in a more intelligent and automatic manner.



TECHNICAL BENEFIT

As illustrated in the following figure, FogFlow provides a standard-based and data-centric edge programming model for IoT service providers to easily and fast realize their services for various business demands. With its data-driven and optimized service orchestration mechanism, FogFlow helps infrastructure providers to automatically and efficiently manage thousands of cloud and edge nodes for city-scale IoT services to achieve optimized performance. In large scale IoT projects like Smart Cities or Smart Factories, FogFlow can therefore save development and operation cost, improve productivity, provide fast time-to-market, as well as increase scalability and reliability.



DIFFERENTIATION

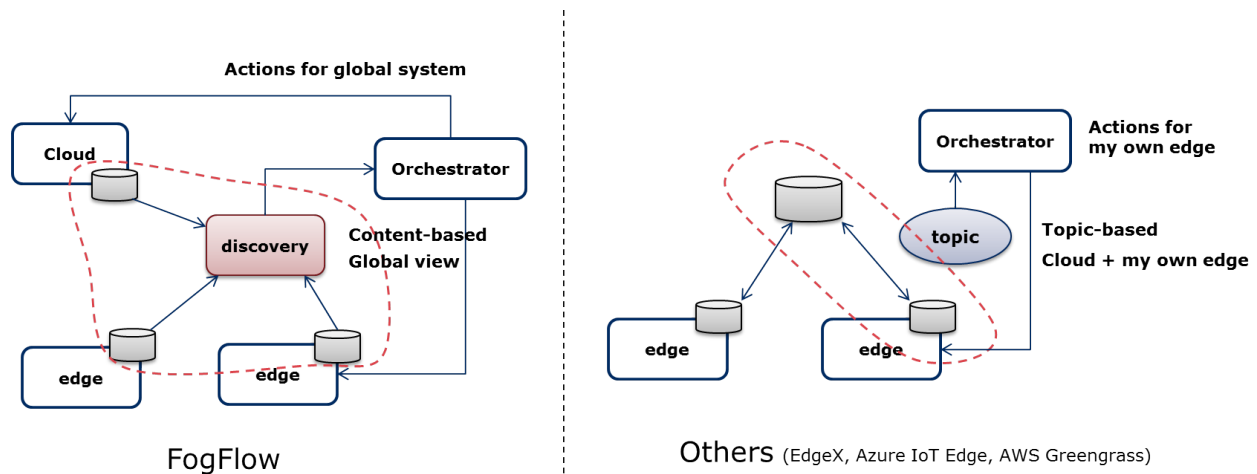
As compared to the other existing IoT edge computing frameworks, such as EdgeX, Azure IoT Edge, Amazon Greengrass. FogFlow has the following unique features illustrated in the following picture.

- **The service orchestration in FogFlow is driven by context, rather than raw events or topics.**

This feature is enabled by the design of introducing a new layer, namely IoT Discovery, which provides a update summary of available entity data on all brokers. As compared to event or topic based orchestration, our context-based orchestration in FogFlow is more flexible and more light-weight. This is because the orchestration decisions in FogFlow can be made based on aggregated context, without reading through all involved data streams. On the other hand, FogFlow takes into account the high level intentions defined by users to make optimized orchestration decisions for better QoS.

- **The FogFlow services and applications are designed against a global view of all cloud nodes and edge nodes, rather than from the perspective of each individual edge node.**

This design principle can largely simplify the required development effort and management overhead, especially FogFlow can support well distributed applications which could run across all cloud nodes and edge nodes seamlessly without knowing the details of how tasks coordination between cloud and edge or between different edge nodes should be carried out. However, for most of the other IoT Edge Computing frameworks, services or applications are designed for each edge and they are not really distributed services or applications, because those services or applications are able to run either in the cloud or at some edge but they are not able to run over cloud nodes and edge nodes in a distributed fashion.



More detailed differentiations are summarized in the following table.

Systems	FogFlow	Others (EdgeX, Azure IoT Edge/AWS Greengrass)
Triggering-mechanism	Content-based	Topic-based
View for orchestration	Global (all edges + cloud)	Each edge + backup broker in the cloud
Programming model(s)	Context-driven functions (Service topology + fog function)	topic-driven Function
Mobility support	Yes	No

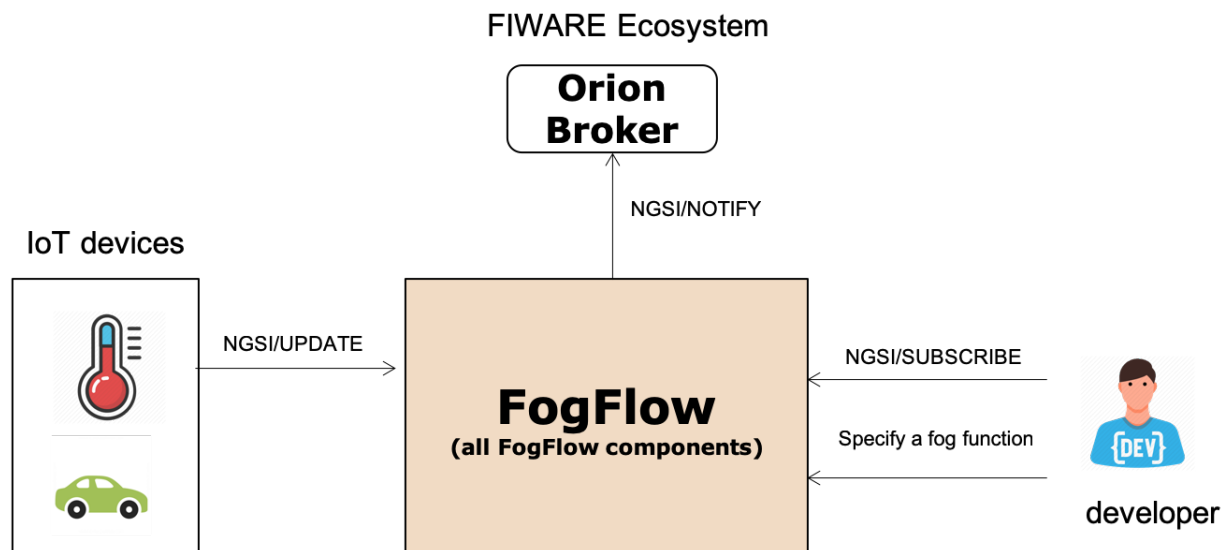
QUICK START

This is an one-page introductory tutorial to FogFlow. In the FIWARE-based architecture, FogFlow can be used to dynamically trigger data processing functions between IoT devices and Orion Context Broker, for the purpose of transforming and preprocessing raw data at edge nodes (e.g., IoT gateways or Raspberry Pis).

The tutorial introduces a typical FogFlow system setup with a simple example to do anomaly detection at edges for temperature sensor data. It explains an example usecase implementation using FogFlow and FIWARE Orion in integration with each other.

Every time to implement a usecase FogFlow creates some internal NGSI entities such as operators, Docker image, Fog Function and service topology. So these entity data are very important for FogFlow system and these are need to be stored somewhere. An entity data can not be stored in FogFlow memory because memory is volatile and it will lose content when power is lost. To solve this issue FogFlow introduces **Dgraph** a persistent storage. The persistent storage will store FogFlow entity data in the form of graph.

As shown in the following diagram, in this use case a connected temperature sensor sends an update message to the FogFlow system, which triggers some running task instance of a pre-defined fog function to generate some analytics result. The fog function is specified in advance via the FogFlow dashboard, however, it is triggered only when the temperature sensor joins the sytem. In a real distributed setup, the running task instance will be deployed at the edge node closed to the temperature sensor. Once the generated analytics result is generated, it will be forwarded from the FogFlow system to Orion Context Broker. This is because a subscription with Orion Context Broker as the reference URL has been issued.



Here are the prerequisite commands for running FogFlow:

1. docker

2. docker-compose

For ubuntu-16.04, you need to install docker-ce and docker-compose.

To install Docker CE, please refer to [Install Docker CE](#), required version > 18.03.1-ce;

Important: please also allow your user to execute the Docker Command without Sudo

To install Docker Compose, please refer to [Install Docker Compose](#), required version 18.03.1-ce, required version > 2.4.2

5.1 Fetch all required scripts

Download the docker-compose file and the configuration files as below.

```
# the docker-compose file to start all FogFlow components on the cloud node
wget https://raw.githubusercontent.com/smartfog/fogflow/master/release/3.2.8/cloud/
↪docker-compose.yml

# the configuration file used by all FogFlow components
wget https://raw.githubusercontent.com/smartfog/fogflow/master/release/3.2.8/cloud/
↪config.json
```

5.2 Change the IP configuration accordingly

You need to change the following IP addresses in config.json according to your own environment and also check if the used port nubmers are blocked by your firewall.

- **my_hostip:** this is the IP of your host machine, which should be accessible for both the web browser on your host machine and docker containers. Please DO NOT use “127.0.0.1” for this.
- **site_id:** each FogFlow node (either cloud node or edge node) requires to have a unique string-based ID to identify itself in the system;
- **physical_location:** the geo-location of the FogFlow node;
- **worker.capacity:** it means the maximal number of docker containers that the FogFlow node can invoke;

Important: please DO NOT use “127.0.0.1” as the IP address of **my_hostip**, because it is only accessible to a running task inside a docker container.

Firewall rules: To make FogFlow web portal accessible and for its proper functioning, the following ports must be free and open over TCP in host machine.

Component	Port
Discovery	8090
Broker	8070
Designer	8080
Nginx	80
Rabbitmq	5672
Task	launched over any port internally

Note : Task Instance is launched over dynamically assigned port, which is not predefined. So, users can possibly allow local ports using rule in his firewall. This will result in smooth functioning of Task Instances.

Above mentioned port number(s) are default port number(s). If user needs to change the port number(s), please make sure the change is consistence in all the configuration files named as “**config.json**”.

Mac Users: if you like to test FogFlow on your Macbook, please install Docker Desktop and also use “host.docker.internal” as my_hostip in the configuration file.

5.3 Start all Fogflow components

Pull the docker images of all FogFlow components and start the FogFlow system

```
#if you already download the docker images of FogFlow components, this command can fetch
the updated images
docker-compose pull

docker-compose up -d
```

5.4 Validate your setup

There are two ways to check if the FogFlow cloud node is started correctly:

- Check all the containers are Up and Running using “docker ps -a”

```
docker-compose ps

795e6afe2857   nginx:latest           "/docker-entrypoint..."   About a minute ago
↳ Up About a minute   0.0.0.0:80->80/tcp
↳ fogflow_nginx_1
33aa34869968   fogflow/worker:3.2.8    "/worker"                   About a minute ago
↳ Up About a minute
↳ fogflow_cloud_worker_1
e4055b5cdfe5   fogflow/master:3.2.8    "/master"                   About a minute ago
↳ Up About a minute   0.0.0.0:1060->1060/tcp
↳ fogflow_master_1
cdf8d4068959   fogflow/designer:3.2.8  "node main.js"             About a minute ago
↳ Up About a minute   0.0.0.0:1030->1030/tcp, 0.0.0.0:8080->8080/tcp
↳ fogflow_designer_1
56daf7f078a1   fogflow/broker:3.2.8    "/broker"                   About a minute ago
↳ Up About a minute   0.0.0.0:8070->8070/tcp
↳ fogflow_cloud_broker_1
51901ce6ee5f   fogflow/discovery:3.2.8 "/discovery"               About a minute ago
↳ Up About a minute   0.0.0.0:8090->8090/tcp
↳ fogflow_discovery_1
eb31cd255fde   rabbitmq:3              "docker-entrypoint.s..."   About a minute ago
↳ Up About a minute   4369/tcp, 5671/tcp, 15691-15692/tcp, 25672/tcp, 0.0.0.0:5672->5672/
↳ tcp
↳ fogflow_rabbitmq_1
```

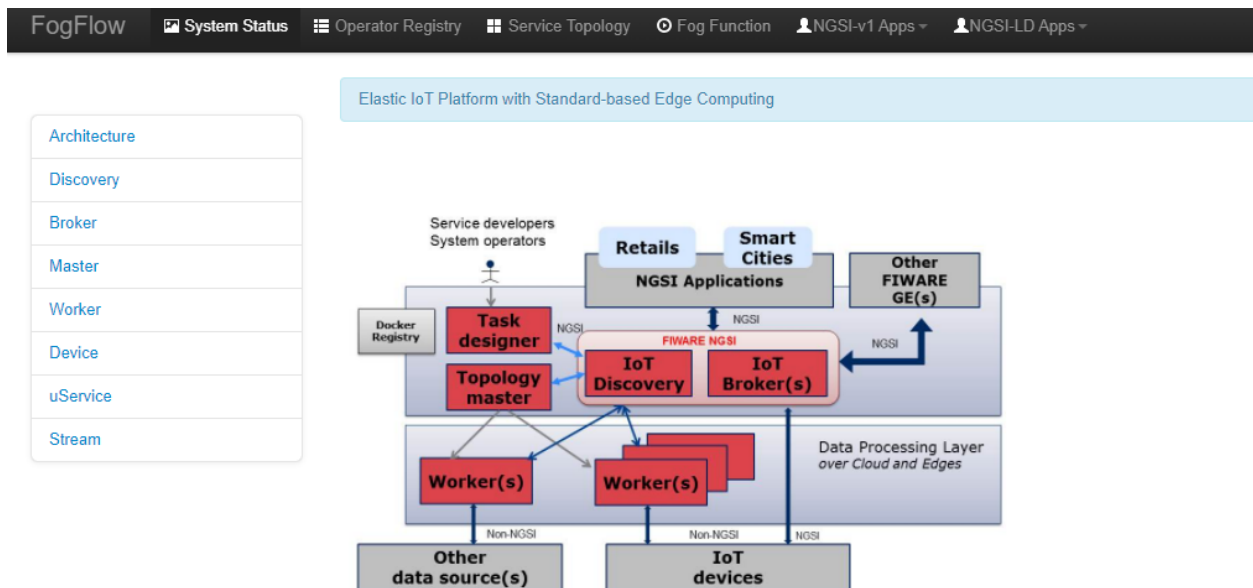
Important: if you see any container is missing, you can run “docker ps -a” to check if any FogFlow component is terminated with some problem. If there is, you can further check its output log by running “docker logs [container ID]”

- Check the system status from the FogFlow DashBoard

You can open the FogFlow dashboard in your web browser to see the current system status via the URL: http://<my_hostip>/index.html

Important: If the FogFlow cloud node is behind a gateway, you need to create a mapping from the gateway IP to the my_hostip and then access the FogFlow dashboard via the gateway IP; If the FogFlow cloud node is a VM in a public cloud like Azure Cloud, Google Cloud, or Amazon Cloud, you need to access the FogFlow dashboard via the public IP of your VM;

Once you are able to access the FogFlow dashboard, you can see the following web page



HELLO WORLD EXAMPLE

Once the FogFlow cloud node is set up, you can try out some existing IoT services without running any FogFlow edge node. For example, you can try out a simple fog function as below.

6.1 Initialize all defined services with three clicks

- Click “Operator Registry” in the top navigator bar to trigger the initialization of pre-defined operators.

After you first click “Operator Registry”, a list of pre-defined operators will be registered in the FogFlow system. With a second click, you can see the refreshed list as shown in the following figure.

The screenshot shows the FogFlow web interface. The top navigation bar includes links for System Status, Operator Registry (active), Service Topology, Fog Function, and NGSI-v1/v2 Apps. The main content area is titled "list of all registered operators" and features a "register" button. Below this is a table listing various operators and their parameters.

Operator	Description	#Parameters
nodejs		0
python		0
iotagent		0
counter		0
anomaly		0
facefinder		0
connectedcar		0
recommender		0
privatesite		0
publicsite		0
pushbutton		0
acoustic		0
speaker		0
dummy		0
geohash		0

- Click “Service Topology” in the top navigator bar to trigger the initialization of pre-defined service topologies.

After you first click “Service Topology”, a list of pre-defined topologies will be registered in the FogFlow system. With a second click, you can see the refreshed list as shown in the following figure.

- Click “Fog Function” in the top navigator bar to trigger the initialization of pre-defined fog functions.

After you first click “Fog Function”, a list of pre-defined functions will be registered in the FogFlow system. With a second click, you can see the refreshed list as shown in the following figure.

FogFlow System Status Operator Registry **Service Topology** Fog Function NGSI-v1 Apps NGSI-LD Apps

list of all registered service topologies

Service Topology

Service Intent

Task Instance

register

Name	Description	#Tasks	Actions
anomaly-detection	detect anomaly events in shops	2	view delete
anomaly-detection.Id	detect anomaly events in shops	2	view delete
child-finder	search for a lost child based on face recognition	1	view delete
Crop_Prediction	This is a ML based approach for crop selection in farming	1	view delete
Heart_Health_Predictor	ML based health prediction of Human Heart	2	view delete
Id-child-finder		1	view delete

FogFlow System Status Operator Registry Service Topology **Fog Function** NGSI-v1 Apps NGSI-LD Apps

list of all registered fog functions

Fog Function

Task Instance

register

ID	Name	Action	Topology
FogFunction.Convert3	Convert3	view delete	{ "name": "Convert3", "description": "test", "tasks": [{ "name": "Main", "operator": "converter", "input_streams": [{ "selected_type": "ConnectedCar", "selected_attributes": [], "groupby": "ALL", "scoped": false }], "output_streams": [{ "entity_type": "RainObservation" }] }] }
FogFunction.Convert2	Convert2	view delete	{ "name": "Convert2", "description": "test", "tasks": [{ "name": "Main", "operator": "geohash", "input_streams": [{ "selected_type": "SmartAwning", "selected_attributes": [], "groupby": "ALL", "scoped": false }], "output_streams": [] }] }
FogFunction.Prediction	Prediction	view delete	{ "name": "Prediction", "description": "test", "tasks": [{ "name": "Main", "operator": "predictor", "input_streams": [{ "selected_type": "RainObservation", "selected_attributes": [], "groupby": "ALL", "scoped": false }], "output_streams": [{ "entity_type": "Prediction" }] }] }
FogFunction.Prediction	Prediction	view delete	{ "name": "Prediction", "description": "test", "tasks": [{ "name": "Main", "operator": "predictor", "input_streams": [{ "selected_type": "RainObservation", "selected_attributes": [], "groupby": "ALL", "scoped": false }], "output_streams": [{ "entity_type": "Prediction" }] }] }

6.2 Simulate an IoT device to trigger the Fog Function

There are two ways to trigger the fog function:

1. Create a “Temperature” sensor entity via the FogFlow dashboard

You can register a device entity via the device registration page: “System Status” -> “Device” -> “Add”. Then you can create a “Temperature” sensor entity by filling the following element: - **Device ID**: to specify a unique entity ID - **Device Type**: use “Temperature” as the entity type - **Location**: select a location on the map

2. Send an NGSI entity update to create the “Temperature” sensor entity

Send a curl request to the FogFlow broker for entity update:

```
curl -iX POST \
  'http://my_hostip/ngsi10/updateContext' \
  -H 'Content-Type: application/json' \
  -d '
{
  "contextElements": [
    {
      "entityId": {
        "id": "Device.Temp001",
        "type": "Temperature",
        "isPattern": false
      },
      "attributes": [
        {
          "name": "temperature",
          "type": "float",
          "value": 73
        },
        {
          "name": "pressure",
```

(continues on next page)

(continued from previous page)

```

        "type": "float",
        "value": 44
      }
    ],
    "domainMetadata": [
      {
        "name": "location",
        "type": "point",
        "value": {
          "latitude": -33.1,
          "longitude": -1.1
        }
      }
    ]
  },
  "updateAction": "UPDATE"
}'

```

6.3 Check if the fog function is triggered

Check if a task is created under “Task” in System Management.**

ID	Service	Task	Type	Worker	port	status
Task.826344878	Task	Dummy	Dummy	Worker.001	40929	running

Check if a Stream is created under “Stream” in System Management.**

ID	Entity Type	Attributes
Result.Temperature.temp001	Result	{\"DeviceID\": {\"type\": \"string\", \"value\": \"temp001\"}, \"url\": {\"type\": \"string\", \"value\": \"http://designer:8080/photo/null\"}, \"iconURL\": {\"type\": \"string\", \"value\": \"photo/default/icon.png\"}}

In this part of the document a conceptual overview of FogFlow and how to use FogFlow in developing any instance is being covered. FogFlow is a cloud and edge environment to orchestrate dynamic **NGSI**-based (Next Generation Service Interface - based) data processing flows on-demand between producers and consumers for providing timely results to make fast actions. A context producer will be a sensor based device whereas a consumer is an Actuator device that will receive command to perform some action.

FogFlow can carry out IoT service orchestration decisions in a decentralized and autonomous manner. This means each FogFlow edge node can make its own decisions only based on a local context view. This way the majority of workloads can be directly handled at edges without always relying on the central cloud. With this “cloudless” approach, FogFlow can not only provide fast response time, but also achieve high scalability and reliability.

To define and trigger FogFlow based instances refer [Intent based programming model](#) part of this document.

CORE CONCEPTS

7.1 Operator

In FogFlow an operator presents a type of data processing unit, which receives certain input streams as NGSI10 notify messages via a listening port, processes the received data, generates certain results, and publishes the generated results as NGSI10 updates.

The implementation of an operator is associated with at least one docker images. To support various hardware architectures (e.g., X86 and ARM for 64bits or 32 bits), the same operator can be associated with multiple docker images.

7.2 Task

A task is a data structure to represent a logic data processing unit within a service topology. Each task is associated with an operator. A task is defined with the following properties:

- **name:** a unique name to present this task
- **operator:** the name of its associated operator
- **groupBy:** the granularity to control the unit of its task instances, which is used by service orchestrator to determine how many task instances must be created
- **input_streams:** the list of its selected input streams, each of which is identified by an entity type
- **output_streams:** the list of its generated output streams, each of which is identified by an entity type

In FogFlow, each input/output stream is represented as a type of NGSI context entities, which are usually generated and updated by either an endpoint device or a data processing task.

During the runtime, multiple task instances can be created for the same task, according to its granularity defined by the *groupBy* property. In order to determine which input stream goes to which task instances, the following two properties are introduced to specify the input streams of tasks:

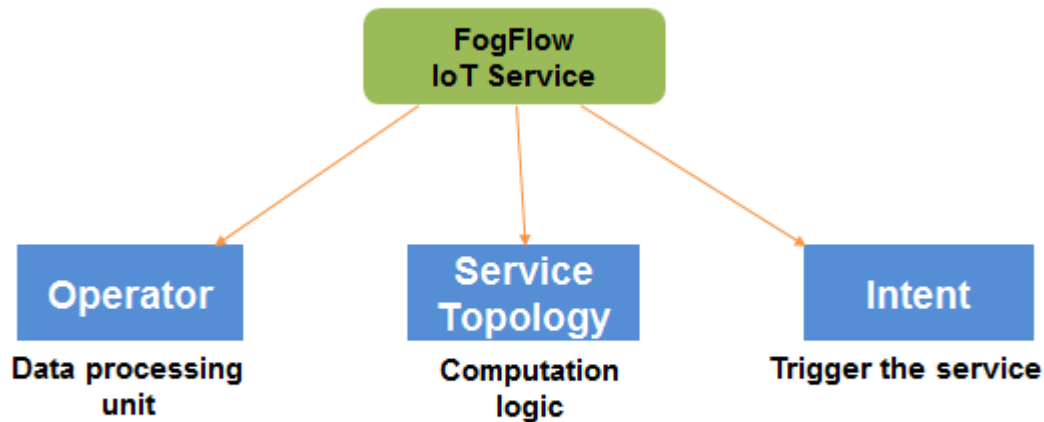
- **Shuffling:** associated with each type of input stream for a task; its value can be either *broadcast* or *unicast*.
 - **broadcast:** the selected input streams should be repeatedly assigned to every task instance of this operator
 - **unicast:** each of the selected input streams should be assigned to a specific task instance only once
- **Scoped:** determines whether the geo-scope in the requirement should be applied to select the input streams; its value can be either *true* or *false*.

7.3 Task Instance

During the runtime, a task is configured by FogFlow with its input data and specified output type and then the configured task will be launched as a task instance, running in a docker container. Currently, each task instance is deployed in a dedicated docker container, either in the cloud or at an edge node.

7.4 IoT Service

The three key elements to program an IoT service is illustrated via below figure.

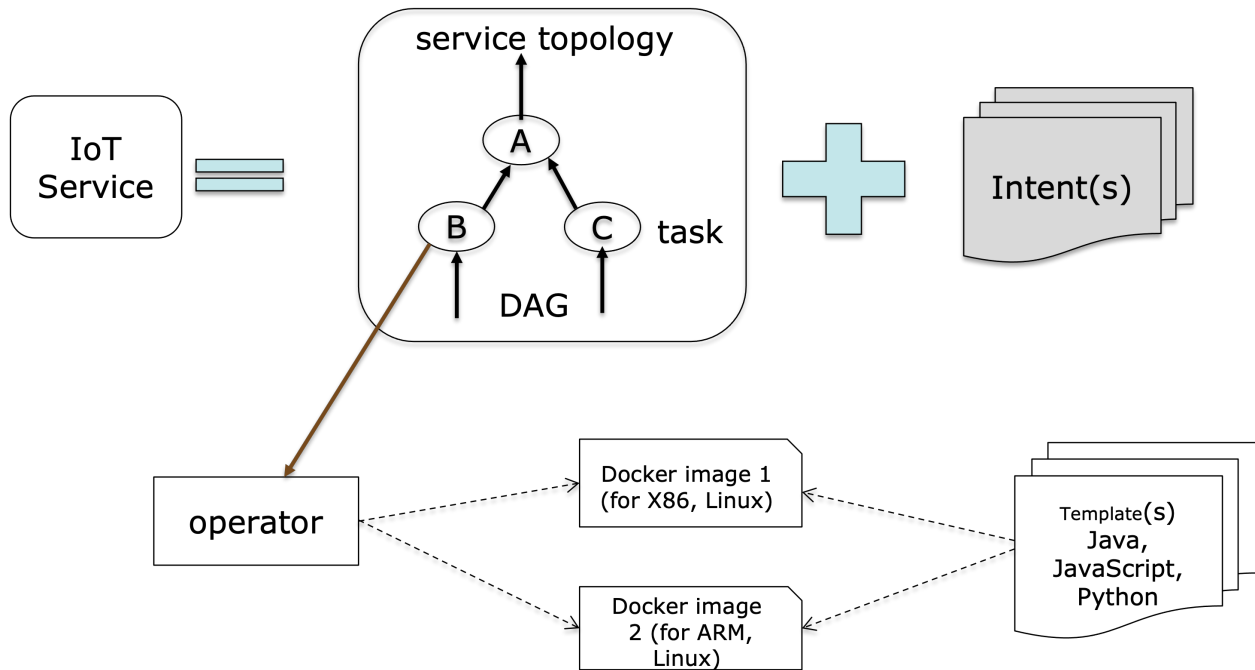


In FogFlow several operators form a graph which is defined as a service topology. Each operator in the service topology is annotated with its inputs and outputs, which indicate their dependency to the other tasks in the same topology. Service topology can easily compose different operators to form their service logic in just a few minutes. After that, during the runtime data processing flows can be automatically triggered based on the high level data usage intent defined by service users. Service users can be either data producers or result consumers.

7.5 Service Topology

Each IoT service is described by a service topology, which is a directed acyclic graph (DAG) to link multiple operators together according to their data dependency. For example, when a service topology is used to specify the service topology, the following information will be included.

- topology name: the unique name of the topology
- service description: some text to describe what this service is about
- priority: define the priority level of all tasks in the topology, which will be utilized by edge nodes to decide how resource should be assigned to tasks
- resource usage: define if the tasks in this topology can use the resources on edge nodes in an exclusive way, meaning that not sharing resources with any task from the other topologies

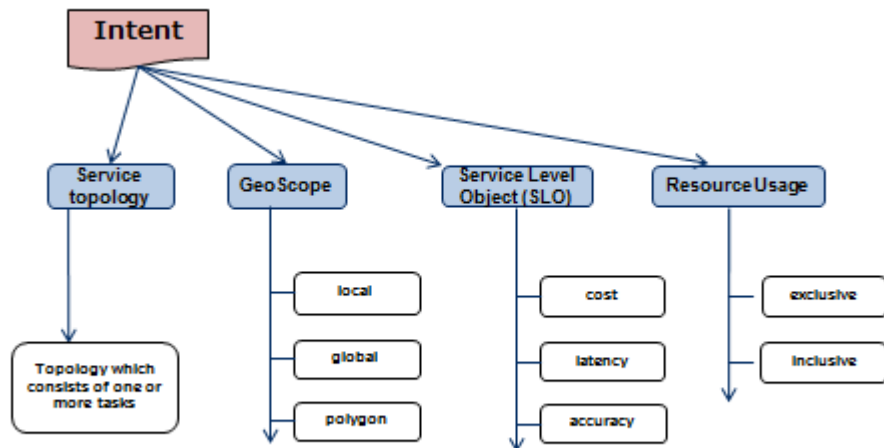


Currently, FogFlow provides a graphical editor to allow developers to easily define and annotate their service topology or fog function during the design phrase.

7.6 Service Intent

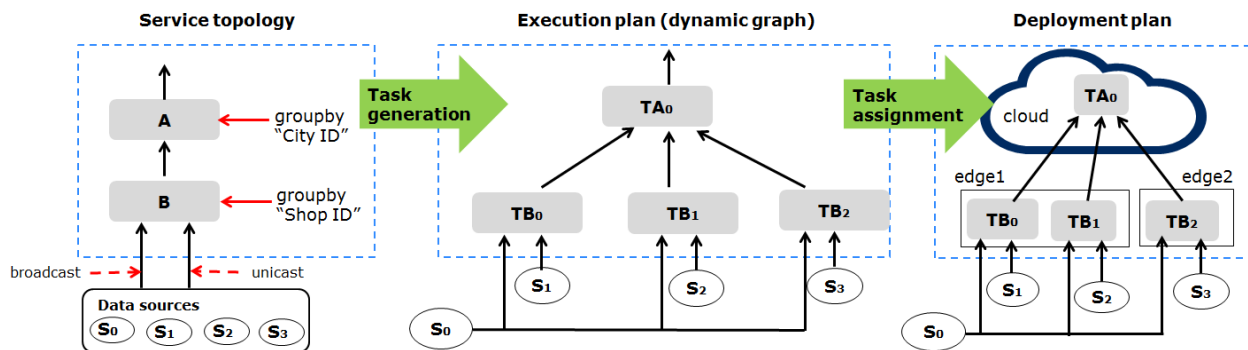
A service intent defines how to trigger an existing service topology. It basically consists of these properties as illustrated in below figure.

- Service topology specifies the computation logic for which the intent object is triggered.
- Geoscope is defined a geographical location where input streams should be selected. Geoscope can be selected as global value as well as can be set custom geoscopes.
- Service Level Object (SLO) is the objective of maximum throughput, minimum latency and minimum cost can be set for task assignment at workers. However, this feature is not fully supported yet, so User can ignore this. It can be set as “None” for now.
- Resource Usage defines how a topology can use resources on edge nodes. It can either exclusive or inclusive. In an exclusive way means the topology will not share the resources with any task from other topologies. Whereas an inclusive topology will share the resources with any task from other topologies.



7.7 From Service Topology to Actual Execution

On receiving a requirement, Topology Master creates a dataflow execution graph and then deploys them over the cloud and edges. The main procedure is illustrated by the following figure, including two major steps.

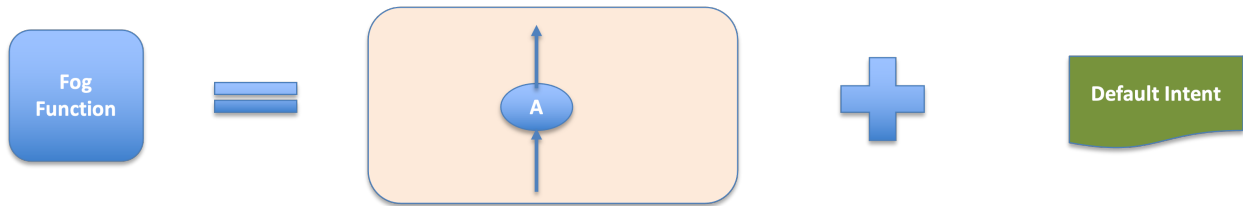


- **from service topology to execution plan: done by the task generation algorithm of Topology Master.**
The generated execution plan includes: 1) which part of service topology is triggered; 2) how many instances need to be created for each triggered task; 3) and how each task instance should be configured with its input streams and output streams.
- **from execution plan to deployment plan: done by the task assignment algorithm of Topology Master.**
The generated deployment plan determines which task instance should be assigned to which worker (in the cloud or at edges), according to certain optimization objectives. Currently, the task assignment in FogFlow is optimized to reduce across-node data traffic without overloading any edge node.

7.8 Fog Function

Currently, FogFlow can support serverless fog computing by providing so-called Fog Function, which is a common easy case of the intent-based programming model. As illustrated in the figure below, Fog Function represents a common special case of the generic intent-based programming model in FogFlow, meaning that a fog function is associated with a simple service topology that includes only one task (a task is mapped to an operator in FogFlow) and an default intent that takes “global” as its geoscope. Therefore, when a fog function is submitted, its service topology will be triggered immediately once its required input data is available.

service topology with a single task



PROGRAMMING MODEL

Currently the following two programming models are provided by FogFlow to support different types of workload patterns.

1. **Fog Function**
2. **Service Topology**

8.1 Fog Function

8.1.1 Define and trigger a fog function

FogFlow enables serverless edge computing, meaning that developers can define and submit a so-called fog function and then the rest will be done by FogFlow automatically, including:

- triggering the submitted fog function when its input data are available
- deciding how many instances to be created according to its defined granularity
- deciding where to deploy the created instances

The instances in the above text refer to the task instances which run a processing logic within them and this processing logic is given by operators in fogflow. They must be registered beforehand by the users. Implementation of an example operator is given in the next sections.

8.1.2 Register task operators

Operator code must be in the form of a docker image and must be available on docker hub. Registration of an operator in FogFlow can be done in one of the following two ways.

Note: Please notice that each operator must have a unique name but the same operator can be associated with multiple docker images, each of which is for one specific hardware or operating system but for implementing the same data processing logic. During the runtime, FogFlow will select a proper docker image to run a scheduled task on an edge node, based on the execution environment of the edge node.

8.1.3 Register it via FogFlow Task Designer

There are two steps to register an operator in Fogflow.

Register an Operator to define what would be the name of Operator and what input parameters it would need. Here in this context, an operator is nothing but a named element having some parameters. The following picture shows the list of all registered operators and their parameter count.

The screenshot shows the FogFlow interface with the 'Operator Registry' tab selected. A sidebar on the left contains 'Operator' and 'Docker Image' buttons. The main area has a 'list of all registered operators' header and a 'register' button. Below is a table listing operators and their parameter counts.

Operator	Description	#Parameters
nodejs		0
python		0
iotagent		0
counter		0
anomaly		0
facefinder		0
connectedcar		0
recommender		0
privatesite		0
publicsite		0
pushbutton		0
acoustic		0
speaker		0
dummy		0
geohash		0

After clicking the “register” button, a design area can be seen below and an operator can be created and parameters can be added to it. To define the port for the operator application, use “service_port” and give a valid port number as its value. The application would be accessible to the outer world through this port.

The screenshot shows the FogFlow interface with the 'Operator Registry' tab selected. A sidebar on the left contains 'Operator' and 'Docker Image' buttons. The main area has a 'to specify an operator' header and a 'Submit' button. Below is a design area showing an operator being configured with parameters.

```

graph LR
    P2[Parameter#2  
Name: service_port  
Values: 4041] --> O1[Operator#1  
Name: iota  
Description:  
Parameters]
    P3[Parameter#3  
Name: service_port  
Values: 7896] --> O1
  
```

Register a Docker Image and choose Operator to define the docker image and associate an already registered Operator with it.

The following picture shows the list of all registered docker images and the key information of each image.

FogFlow
System Status
Operator Registry
Service Topology
Fog Function
NGSI-v1 Apps
NGSI-LD Apps

Operator
Docker Image

list of docker images in the docker registry

register

Operator	Image	Tag	Hardware Type	OS Type	Prefetched
python	fogflow/python	latest	X86	Linux	false
counter	fogflow/counter	latest	X86	Linux	false
facefinder	fogflow/facefinder	latest	X86	Linux	false
connectedcar	fogflow/connectedcar	latest	X86	Linux	false
iotagent	fiware/iotagent-json	latest	X86	Linux	false
nodejs	fogflow/nodejs	latest	X86	Linux	true
recommender	fogflow/recommender	latest	X86	Linux	false
anomaly	fogflow/anomaly	latest	X86	Linux	false
publicsite	fogflow/publicsite	latest	X86	Linux	false
pushbutton	pushbutton	latest	ARM	Linux	false
acoustic	acoustic	latest	ARM	Linux	false
speaker	speaker	latest	ARM	Linux	false
pushbutton	pushbutton	latest	X86	Linux	false
privatesite	fogflow/privatesite	latest	X86	Linux	false
dummy	fogflow/dummy	latest	X86	Linux	false

After clicking the “register” button, a form can be seen as below. Please fill out the required information and click the “register” button to finish the registration. The form is explained as the following.

- Image: the name of your operator docker image
- Tag: the tag is used to publish the operator docker image; by default it is “latest”
- Hardware Type: the hardware type that the docker image supports, including X86 or ARM (e.g. Raspberry Pi)
- OS Type: the operating system type that the docker image supports; currently this is only limited to Linux
- Operator: the operator name, which must be unique and will be used when defining a service topology
- Prefetched: if this is checked, that means all edge nodes will start to fetch this docker image in advance; otherwise, the operator docker image is fetched on demand, only when edge nodes need to run a scheduled task associated with this operator.

Important: Please notice that the name of the docker image must be consistent with the one that is published to [Docker Hub](#). By default, FogFlow will fetch the required docker images from Docker Hub using the name that is registered here for an operator.

FogFlow

System Status
Operator Registry
Service Topology
Fog Function
Use Cases

Operator

DockerImage

to register a new docker image

Image(*)

Tag(*)

latest

HardwareType(*)

X86

OSType(*)

Linux

Operator(*)

nodejs

Prefetched

☐ docker image must be fetched by the platform in advance

Register

8.1.4 Register it programmatically by sending a NGSI update

An operator can also be registered by sending docker image for a constructed NGSI update message to the IoT Broker deployed in the cloud.

Here are the Curl and the Javascript-based code examples to register an operator and a docker image for that operator.

Note: In the Javascript code example, Javascript-based library is used to interact with FogFlow IoT Broker. It can be found out in library from the github code repository ([designer/public/lib/ngsi](#)). It shall be included in `ngsiclient.js` of web page.

Note: The Curl case assumes that the cloud IoT Broker is running on localhost on port 8070.

Curl

Javascript

```
curl -iX POST \
  'http://localhost:8070/ngsi10/updateContext' \
-H 'Content-Type: application/json' \
-d '
{
  "contextElements": [
    {
      "entityId":{
        "id":"counter",
        "type":"Operator"
      },

```

(continues on next page)

(continued from previous page)

```

      "attributes": [
        {
          "name": "designboard",
          "type": "object",
          "value": {
          },
        },
        {
          "name": "operator",
          "type": "object",
          "value": {
            "description": "",
            "name": "counter",
            "parameters": [
            ]
          }
        }
      ],
      "domainMetadata": [
        {
          "name": "location",
          "type": "global",
          "value": "global"
        }
      ]
    },
    {
      "entityId": {
        "id": "fogflow/counter.latest",
        "type": "DockerImage"
      },
      "attributes": [
        {
          "name": "image",
          "type": "string",
          "value": "fogflow/counter"
        },
        {
          "name": "tag",
          "type": "string",
          "value": "latest"
        },
        {
          "name": "hwType",
          "type": "string",
          "value": "X86"
        },
        {
          "name": "osType",
          "type": "string",
          "value": "Linux"
        }
      ]
    }
  ]
}

```

(continues on next page)

(continued from previous page)

```

        },
        {
            "name": "operator",
            "type": "string",
            "value": "counter"
        },
        {
            "name": "prefetched",
            "type": "boolean",
            "value": false
        }
    ],
    "domainMetadata": [
        {
            "name": "operator",
            "type": "string",
            "value": "counter"
        },
        {
            "name": "location",
            "type": "global",
            "value": "global"
        }
    ]
},
"updateAction": "UPDATE"
}'

```

```

name = "counter"

//register a new operator
var newOperatorObject = {};

newOperatorObject.entityId = {
    id : name,
    type: 'Operator',
    isPattern: false
};

newOperatorObject.attributes = [];

newOperatorObject.attributes.designboard = {type: 'object', value: {}};

var operatorValue = {}
operatorValue = {description: "Description here...", name: name, parameters: []};
newOperatorObject.attributes.operator = {type: 'object', value: operatorValue};

newOperatorObject.metadata = [];
newOperatorObject.metadata.location = {type: 'global', value: 'global'};

// assume the config.brokerURL is the IP of cloud IoT Broker

```

(continues on next page)

(continued from previous page)

```

var client = new NGSI10Client(config.brokerURL);
client.updateContext(newOperatorObject).then( function(data) {
    console.log(data);
}).catch( function(error) {
    console.log('failed to register the new Operator object');
});

image = {}

image = {
    name: "fogflow/counter",
    tag: "latest",
    hwType: "X86",
    osType: "Linux",
    operator: "counter",
    prefetched: false
};

newImageObject = {};

newImageObject.entityId = {
    id : image.name + '.' + image.tag,
    type: 'DockerImage',
    isPattern: false
};

newImageObject.attributes = [];
newImageObject.attributes.image = {type: 'string', value: image.name};
newImageObject.attributes.tag = {type: 'string', value: image.tag};
newImageObject.attributes.hwType = {type: 'string', value: image.hwType};
newImageObject.attributes.osType = {type: 'string', value: image.osType};
newImageObject.attributes.operator = {type: 'string', value: image.operator};
newImageObject.attributes.prefetched = {type: 'boolean', value: image.prefetched};

newImageObject.metadata = [];
newImageObject.metadata.operator = {type: 'string', value: image.operator};
newImageObject.metadata.location = {type: 'global', value: 'global'};

client.updateContext(newImageObject).then( function(data) {
    console.log(data);
}).catch( function(error) {
    console.log('failed to register the new Docker Image object');
});

```

It is recommended to use fogflow dashboard to create an operator with parameters. However, if the users wish to use curl, then they can refer the following for the example operator registration with parameters shown in the above image. Afterwards, users can register a docker image that uses this operator.

The x and y variables here are simply the coordinates of designer board. If they are not given by user, by default, all the element blocks will be placed at origin of the plane.

```

curl -iX POST \
    'http://localhost:8070/ngsi10/updateContext' \

```

(continues on next page)

(continued from previous page)

```
-H 'Content-Type: application/json' \  
-d '  
{  
  "contextElements": [  
    {  
      "entityId":{  
        "id":"iota",  
        "type":"Operator"  
      },  
      "attributes":[  
        {  
          "name":"designboard",  
          "type":"object",  
          "value":{  
            "blocks":[  
              {  
                "id":1,  
                "module":null,  
                "type":"Parameter",  
                "values":{  
                  "name":"service_port",  
                  "values":[  
                    "4041"  
                  ]  
                },  
                "x":-425,  
                "y":-158  
              },  
              {  
                "id":2,  
                "module":null,  
                "type":"Parameter",  
                "values":{  
                  "name":"service_port",  
                  "values":[  
                    "7896"  
                  ]  
                },  
                "x":-393,  
                "y":-51  
              },  
              {  
                "id":3,  
                "module":null,  
                "type":"Operator",  
                "values":{  
                  "description":"",  
                  "name":"iota"  
                },  
                "x":-186,  
                "y":-69  
              }  
            ]  
          }  
        ]  
      }  
    ]  
  }  
}
```

(continues on next page)

(continued from previous page)

```

    ],
    "edges": [
      {
        "block1": 2,
        "block2": 3,
        "connector1": [
          "parameter",
          "output"
        ],
        "connector2": [
          "parameters",
          "input"
        ],
        "id": 1
      },
      {
        "block1": 1,
        "block2": 3,
        "connector1": [
          "parameter",
          "output"
        ],
        "connector2": [
          "parameters",
          "input"
        ],
        "id": 2
      }
    ]
  },
  {
    "name": "operator",
    "type": "object",
    "value": {
      "description": "",
      "name": "iota",
      "parameters": [
        {
          "name": "service_port",
          "values": [
            "7896"
          ]
        },
        {
          "name": "service_port",
          "values": [
            "4041"
          ]
        }
      ]
    }
  }
]
}

```

(continues on next page)

(continued from previous page)

```

        }
      ],
      "domainMetadata": [
        {
          "name": "location",
          "type": "global",
          "value": "global"
        }
      ]
    },
    "updateAction": "UPDATE"
  }'

```

8.1.5 Define a “Dummy” fog function

The following steps show how to define and test a simple ‘dummy’ fog function using the web portal provided by FogFlow Task Designer. The “dummy” operator is already registered in Fogflow by default.

8.1.6 create a fog function from the FogFlow editor

A menu will pop up whenever click a right mouse on the task design board.

The screenshot shows the FogFlow web portal interface. At the top is a navigation bar with the following items: FogFlow, System Status, Operator Registry, Service Topology, Fog Function (selected), NGSI-v1 Apps, and NGSI-LD Apps. Below the navigation bar is a light blue header with the text "to design a fog function". On the left side, there is a sidebar with two buttons: "Fog Function" (highlighted in blue) and "Task Instance". The main content area contains a form with the following fields:

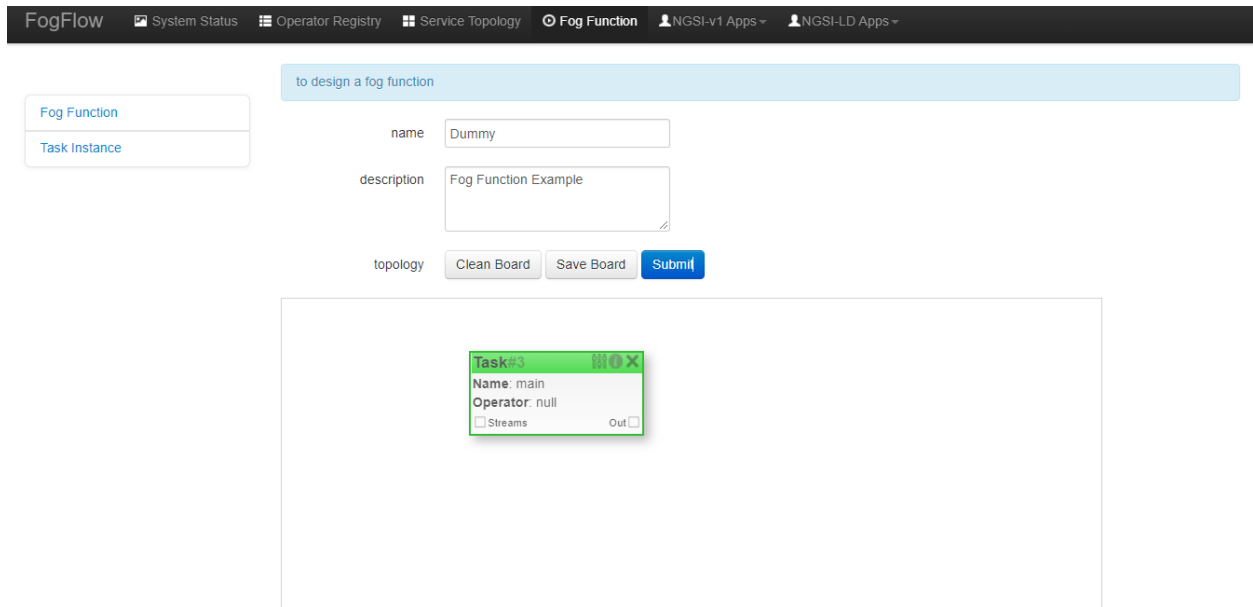
- name:** A text input field containing the word "Dummy".
- description:** A text input field containing the text "Fog Function Example".
- topology:** A section containing three buttons: "Clean Board", "Save Board", and "Submit" (highlighted in blue).

 Below the form is a large rectangular design board. A context menu is open on the design board, showing two options: "Task" and "EntityStream".

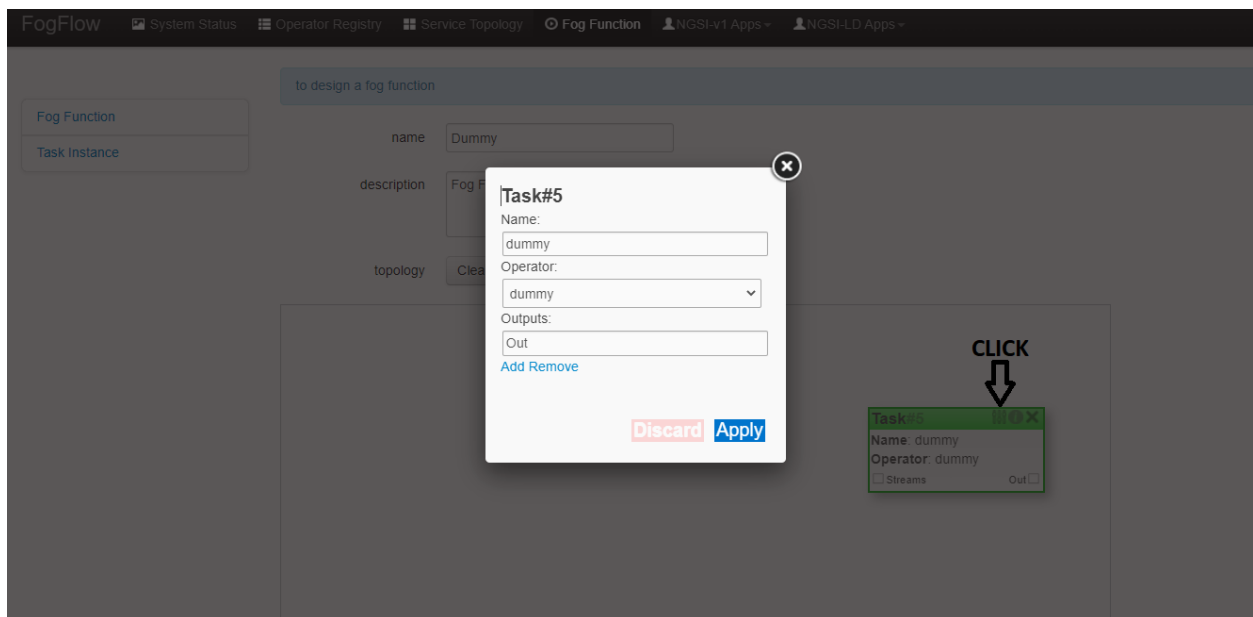
The displayed menu includes the following items:

- **Task:** is used to define the fog function name and the processing logic (or operator). A task has input and output streams.
- **EntityStream:** is the input data element which can be linked with a fog function Task as its input data stream.

Click “Task” from the popup menu, a Task element will be placed on the design board, as shown below.



Start task configuration by clicking the configuration button on the top-right corner, as illustrated in the following figure. Please specify the name of the Task and choose an operator out of a list of some pre-registered operators.



Please click “EntityStream” from the popup menu to place an “EntityStream” element on the design board.

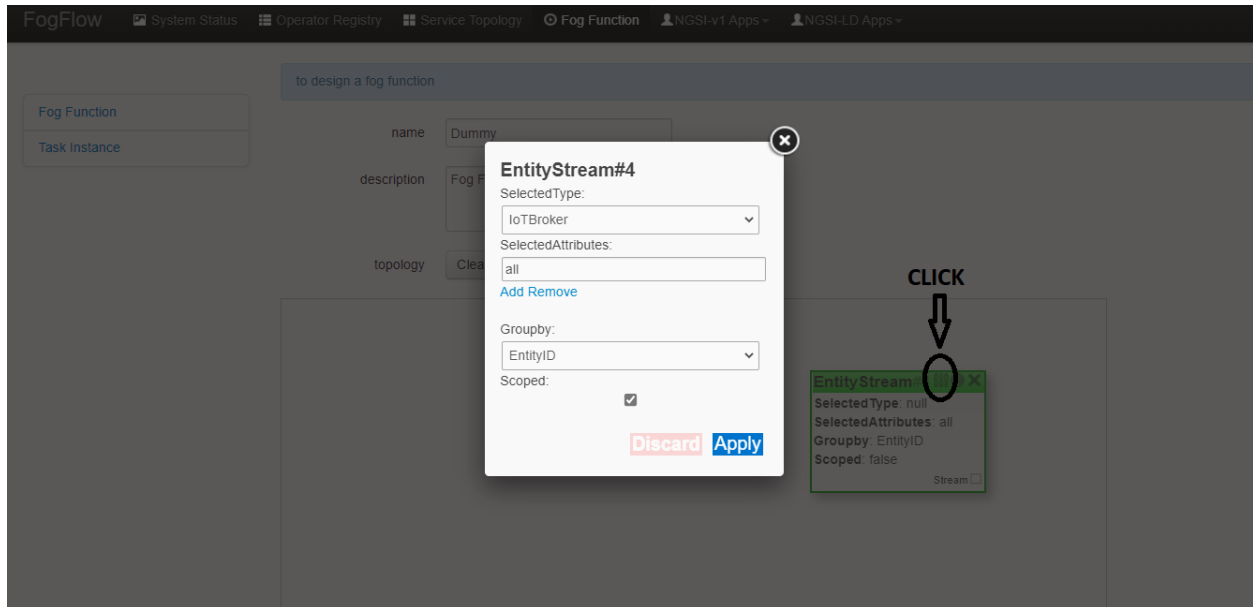
It contains the following things:

- Selected Type: is used to define the entity type of input stream whose availability will trigger the fog function.
- Selected Attributes: for the selected entity type, which entity attributes are required by your fog function; “all” means to get all entity attributes.
- Group By: should be one of the selected entity attributes, which defines the granularity of this fog function.
- Scoped: tells if the Entity data are location-specific or not. True indicates that location-specific data are recorded in the Entity and False is used in case of broadcasted data, for example, some rule or threshold data that holds true for all locations, not for a specific location.

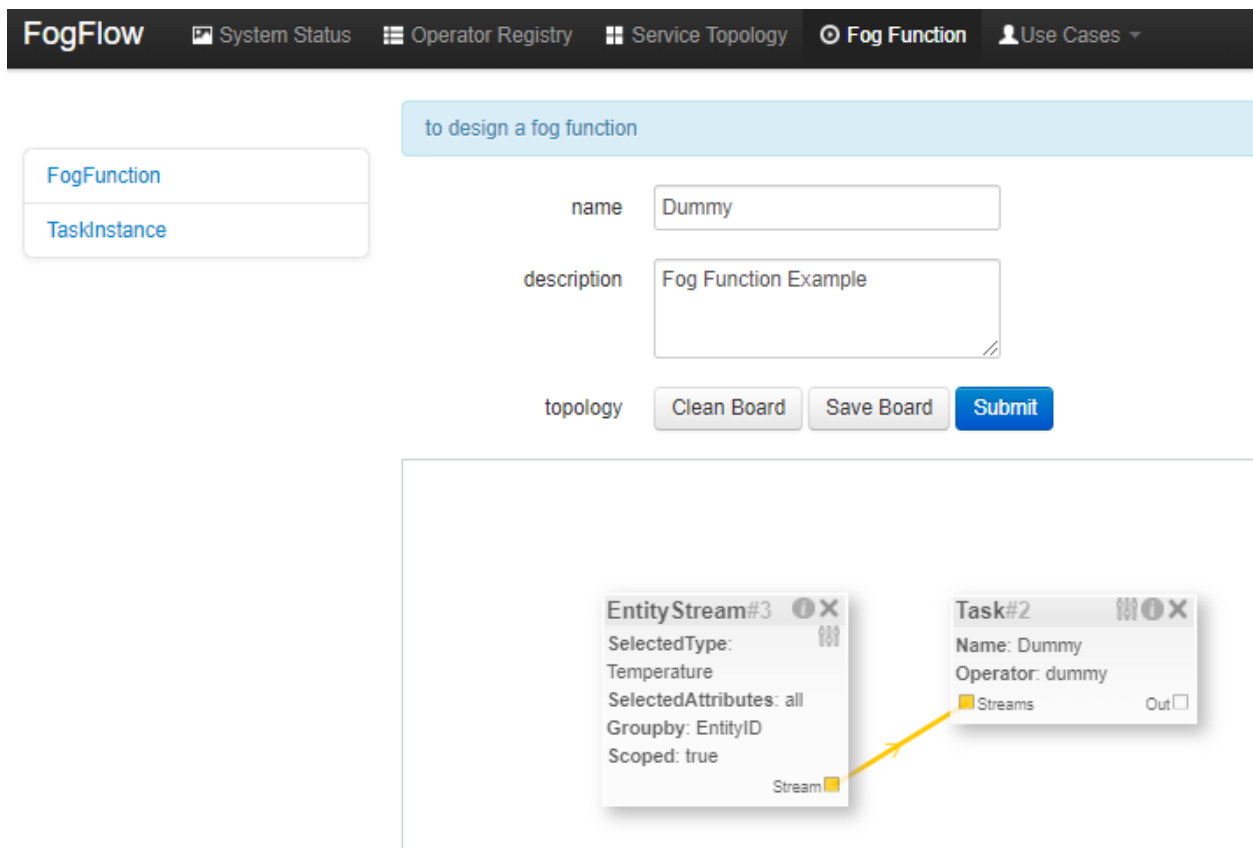
Note: granularity determines the number of instances for this fog function. In principle, the number of task instances for the defined fog function will be equal to the total number of unique values of the selected entity attributes, for the available input data. It also means, each instance will be assigned to handle all input entities with a specific attribute value.

In this example, the granularity is defined by “id”, meaning that FogFlow will create a new task instance for each individual entity ID.

Configure the EntityStream by clicking on its configuration button as shown below. In this example, we choose “Temperature” as the entity type of input data for the “dummy” fog function.



There can be multiple EntityStreams for a Task and they must be connected to the Task as shown here.



8.1.7 provide the code of your own function

Currently FogFlow allows developers to specify their own function code inside a registered operator. For a sample operator, refer the .

```
exports.handler = function(contextEntity, publish, query, subscribe) {
  console.log("enter into the user-defined fog function");

  var entityID = contextEntity.entityId.id;

  if (contextEntity == null) {
    return;
  }
  if (contextEntity.attributes == null) {
    return;
  }

  var updateEntity = {};
  updateEntity.entityId = {
    id: "Stream.result." + entityID,
    type: 'result',
    isPattern: false
  };
  updateEntity.attributes = {};
  updateEntity.attributes.city = {
    type: 'string',
    value: 'Heidelberg'
  };

  updateEntity.metadata = {};
  updateEntity.metadata.location = {
    type: 'point',
    value: {
      'latitude': 33.0,
      'longitude': -1.0
    }
  };

  console.log("publish: ", updateEntity);
  publish(updateEntity);
};
```

Above javascript code example can be taken as the implementation of fog function. This example fog function simple writes a fixed entity by calling the “publish” callback function.

The input parameters of a fog function are predefined and fixed, including:

- **contextEntity**: representing the received entity data
- **publish**: the callback function to publish your generated result back to the FogFlow system
- **query**: optional, this is used only when your own internal function logic needs to query some extra entity data from the FogFlow context management system.
- **subscribe**: optional, this is used only when your own internal function logic needs to subscribe some extra entity data from the FogFlow context management system.

Important: For the callback functions *query* and *subscribe*, “extra” means any entity data that are not defined as the inputs in the annotation of your fog function.

A Javascript-based template of the implementation of fog functions is provided in the FogFlow repository as well. Please refer to [Javascript-based template for fog function](#)

Templates for Java and python are also given in the repository.

Here are some examples to show how these three call back functions can be used.

- **example usage of *publish*:**

```
var updateEntity = {};
updateEntity.entityId = {
  id: "Stream.Temperature.0001",
  type: 'Temperature',
  isPattern: false
};
updateEntity.attributes = {};
updateEntity.attributes.city = {type: 'string', value: 'Heidelberg'};

updateEntity.metadata = {};
updateEntity.metadata.location = {
  type: 'point',
  value: {'latitude': 33.0, 'longitude': -1.0}
};

publish(updateEntity);
```

- **example usage of *query*:**

```
var queryReq = {}
queryReq.entities = [{type:'Temperature', isPattern: true}];
var handleQueryResult = function(entityList) {
  for(var i=0; i<entityList.length; i++) {
    var entity = entityList[i];
    console.log(entity);
  }
}

query(queryReq, handleQueryResult);
```

- **example usage of *subscribe*:**

```
var subscribeCtxReq = {};
subscribeCtxReq.entities = [{type: 'Temperature', isPattern: true}];
subscribeCtxReq.attributes = ['avg'];

subscribe(subscribeCtxReq);
```

8.1.8 submit fog function

After clicking the “Submit” button, the annotated fog function will be submitted to FogFlow.

8.1.9 Trigger “dummy” fog function

The defined “dummy” fog function is triggered only when its required input data are available. With the following command, you can create a “Temperature” sensor entity to trigger the function. Please fill out the following required information:

- **Device ID:** to specify a unique entity ID
- **Device Type:** use “Temperature” as the entity type
- **Location:** to place a location on the map

FogFlow System Status Operator Registry Service Topology Fog Function NGSI-v1 Apps NGSI-LD Apps

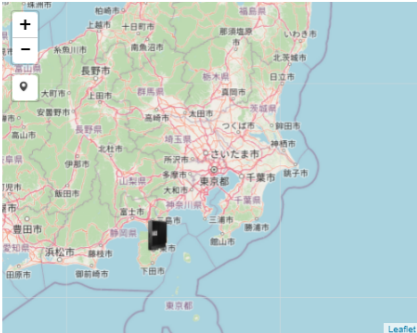
to register a new IoT device

Device ID(*) Autogen

Device Type(*) Temperature

Icon Image Choose File No file chosen

Camera Image Choose File No file chosen

Location(*) 

Register

Once the device profile is registered, a new “Temperature” sensor entity will be created and it will trigger the “dummy” fog function automatically.

FogFlow System Status Operator Registry Service Topology Fog Function Use Cases

list of all IoT devices

add

ID	Type	Attributes	DomainMetadata
Device.Temperature.temp001	Temperature	{ "DeviceID": { "type": "string", "value": "temp001", "url": { "type": "string", "value": "http://designer:8080/photo/null", "iconURL": { "type": "string", "value": "/photo/default/icon.png" } } }	{ "location": { "type": "point", "value": { "latitude": 35.34836246676312, "longitude": 138.96118 } }

Profile Delete

The other way to trigger the fog function is to send a NGSI entity update to create the “Temperature” sensor entity. Following command can be executed to issue a POST request to the FogFlow broker.

```
curl -iX POST \
  'http://localhost:8080/ngsi10/updateContext' \
  -H 'Content-Type: application/json' \
  -d '
{
  "contextElements": [
    {
      "entityId": {
        "id": "Device.temp001",
        "type": "Temperature",
        "isPattern": false
      },
      "attributes": [
        {
          "name": "temp",
          "type": "integer",
```

(continues on next page)

(continued from previous page)

```

        "value": 10
      }
    ],
    "domainMetadata": [
      {
        "name": "location",
        "type": "point",
        "value": {
          "latitude": 49.406393,
          "longitude": 8.684208
        }
      }
    ]
  },
  "updateAction": "UPDATE"
},

```

Check whether the fog function is triggered or not in the following way.

- check the task instance of this fog function, as shown in the following picture

ID	Service	Task	Type	Worker	port	status
Task.826344878	Task	Dummy	Dummy	Worker.001	40929	running

- check the result generated by its running task instance, as shown in the following picture

ID	Entity Type	Attributes
Result.Temperature.temp001	Result	{ "DeviceID": { "type": "string", "value": "temp001", "url": { "type": "string", "value": "http://designer.8080/photo/null", "iconURL": { "type": "string", "value": "photo/default/icon.png" } } } }

8.2 Service Topology

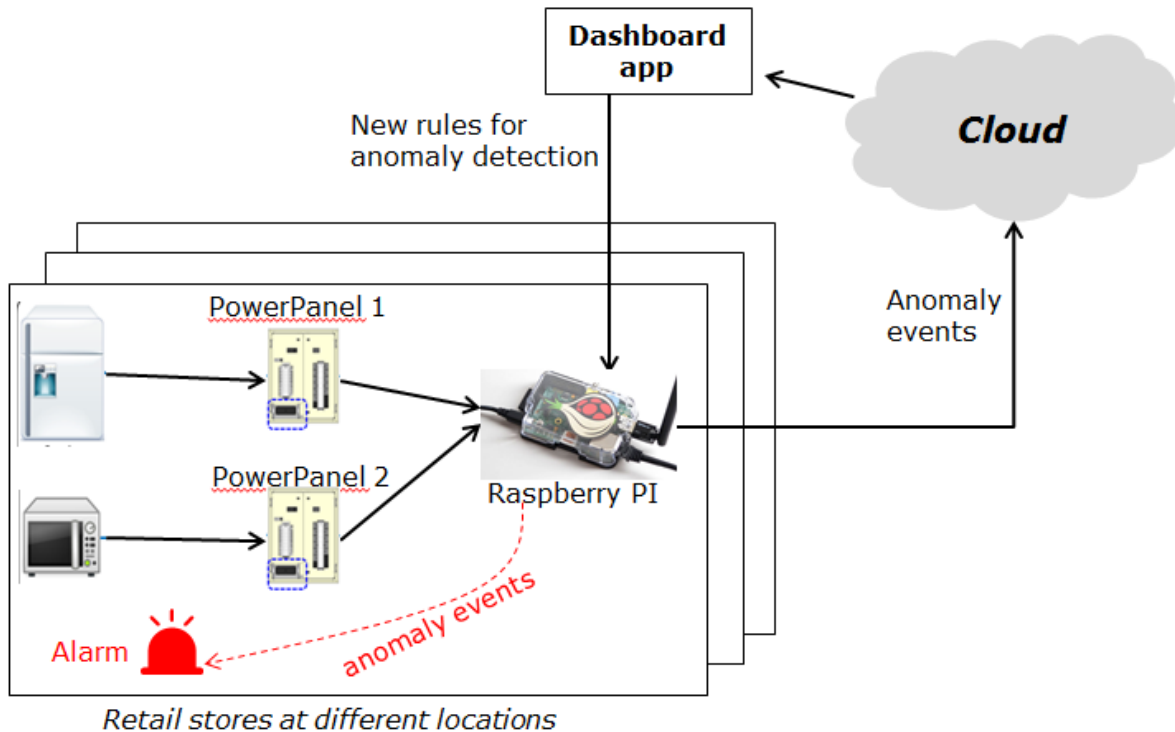
8.2.1 Define and trigger a service topology

In FogFlow a service topology is defined as a graph of several operators. Each operator in the service topology is annotated with its inputs and outputs, which indicate their dependency to the other tasks in the same topology. **Different from fog functions, a service topology is triggered on demand by a customized “intent” object.**

With a simple example we explain how developers can define and test a service topology in the following section.

8.2.2 Use case on anomaly detection

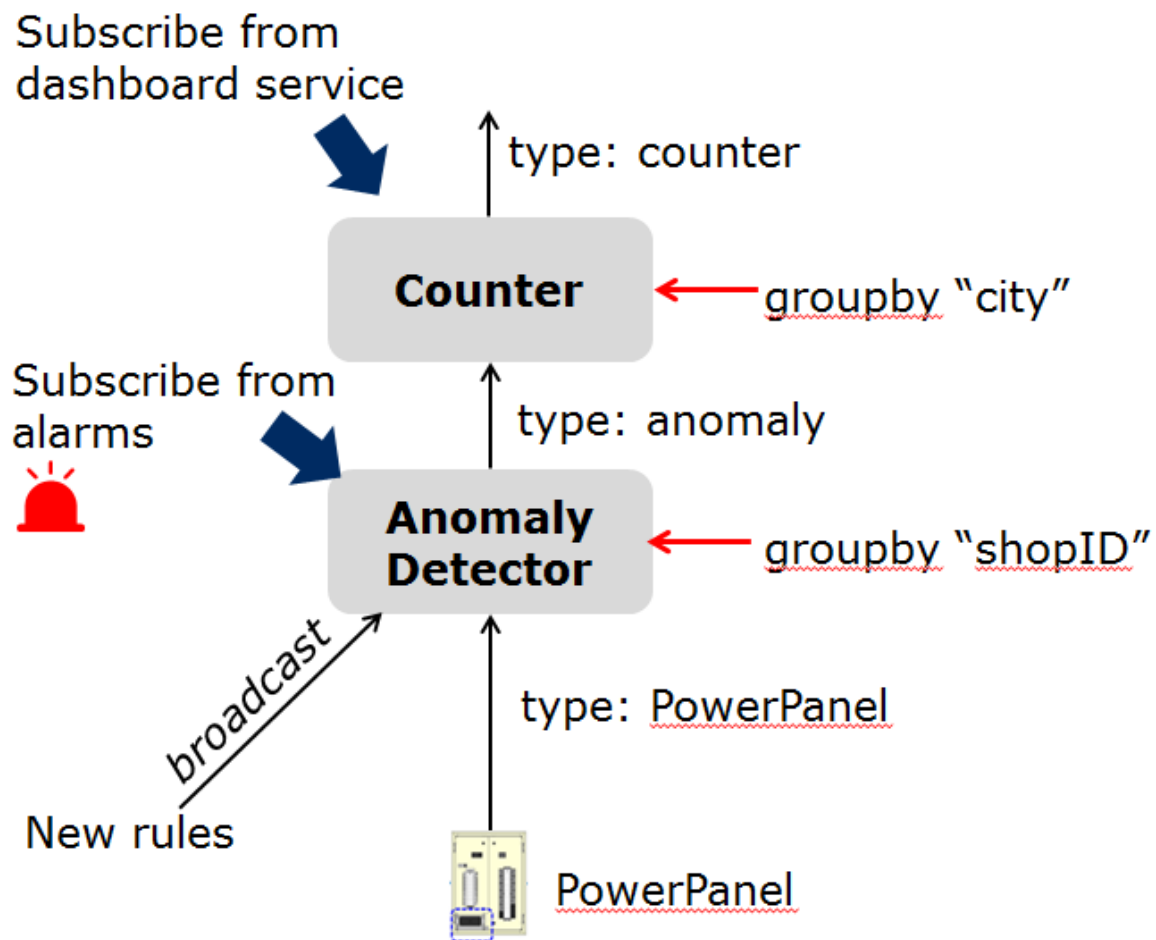
This use case study is for retail stores to detect abnormal energy consumption in real-time. As illustrated in the following picture, a retail company has a large number of shops distributed in different locations. For each shop, a Raspberry Pi device (edge node) is deployed to monitor the power consumption from all PowerPanels in the shop. Once an abnormal power usage is detected on the edge, the alarm mechanism in the shop is triggered to inform the shop owner. Moreover, the detected event is reported to the cloud for information aggregation. The aggregated information is then presented to the system operator via a dashboard service. In addition, the system operator can dynamically update the rule for anomaly detection.



- **Anomaly Detector:** this operator is to detect anomaly events based on the collected data from power panels in a retail store. It has two types of inputs:
 - detection rules, which are provided and updated by the operator; The detection rules input stream type is associated with **broadcast**, meaning that the rules are needed by all task instances of this operator. The granularity of this operator is based on **shopID**, meaning that a dedicated task instance will be created and configured for each shop
 - sensor data from power panel
- **Counter:** this operator is to count the total number of anomaly events for all shops in each city. Therefore, its task granularity is by city. Its input stream type is the output stream type of the previous operator (Anomaly Detector).

There are two types of result consumers:

- (1) a dashboard service in the cloud, which subscribes to the final aggregation results generated by the counter operator for the global scope;
- (2) the alarm in each shop, which subscribes to the anomaly events generated by the Anomaly Detector task on the local edge node in the retail store.



8.2.3 Implement your operator functions required in your service topology

Before you can define the designed service topology, all operators used in your service topology must be provided by you or the other provider in the FogFlow system. For this specific use case, we need to implement two operators: `anomaly_detector` and `counter`. Please refer to the examples provided in our code repository.

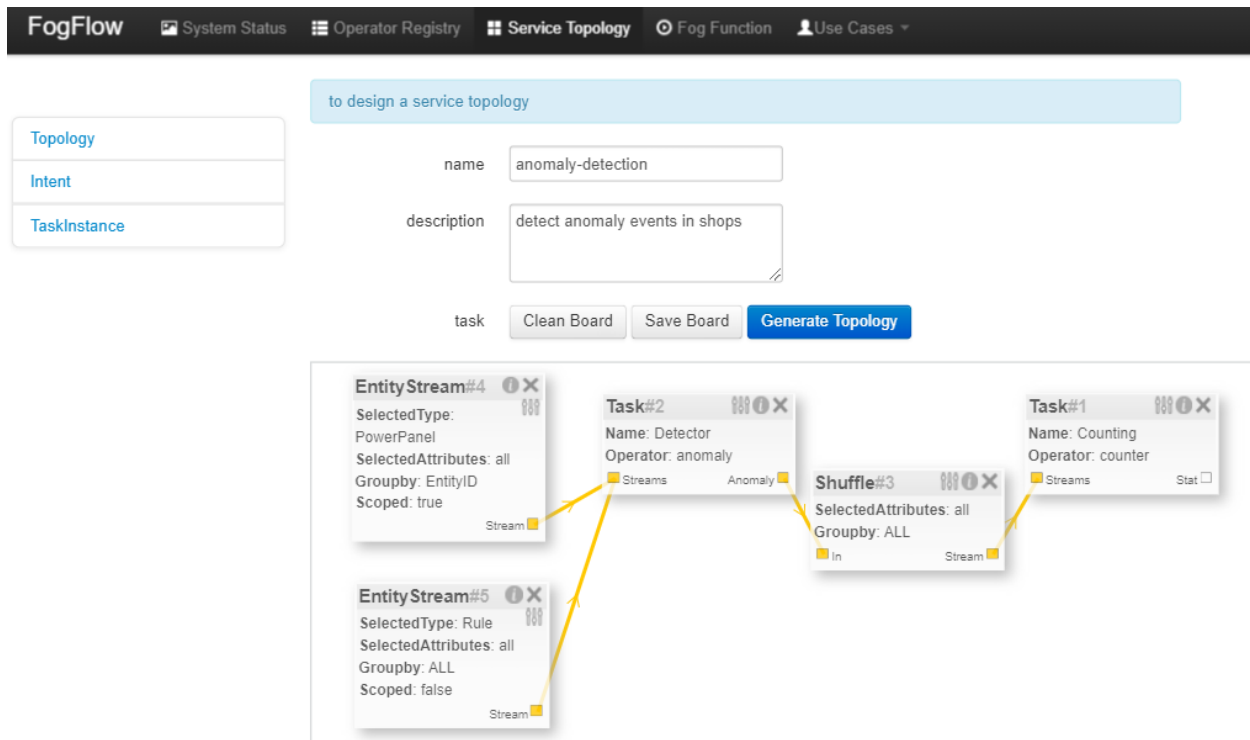
- `anomaly_detector`
- `counter`

8.2.4 Specify a service topology

Assume that the tasks to be used in your service topology have been implemented and registered, you can have two ways to specify your service topology.

8.2.5 using FogFlow Topology Editor

The first way is to use the FogFlow editor to specify a service topology.



As seen in the picture, the following important information must be provided.

1. **define topology profile, including**

- topology name: the unique name of your topology
- service description: some text to describe what this service is about

2. **draw the graph of data processing flows within the service topology**

With a right click at some place of the design board, you will see a menu pops up and then you can start to choose either task or input streams or shuffle to define your data processing flows according to the design you had in mind.

3. define the profile for each element in the data flow, including

As shown in the above picture, you can start to specify the profile of each element in the data processing flow by clicking the configuration button.

The following information is required to specify a task profile.

- name: the name of the task
- operator: the name of the operator that implements the data processing logic of this task; please register your operator beforehand so that it can be shown from the list
- entity type of output streams: to specify the entity type of the produced output stream.

The following information is required to specify an EntityStream Profile.

- SelectedType: is used to define what Entity Type will be chosen by the task as its Input Stream
- SelectedAttributes: is used to define what attribute (or attributes) of the Selected Entity Type will be considered for changing the state of a task.
- Groupby: to determine how many instances of this task should be created on the fly; currently including the following cases
 - if there is only one instance to be created for this task, please use “groupby” = “all”
 - if you need to create one instance for each entity ID of the input streams, please user “groupby” = “entityID”
 - if you need to create one instance for each unique value of some specific context metadata, please use the name of this registered context metadata
- Scoped: tells if the Entity data are location-specific or not. True indicates that location-specific data are recorded in the Entity and False is used in case of broadcasted data, for example, some rule or threshold data that holds true for all locations, not for a specific location.

Shuffling element serves as a connector between two tasks such that output of a task is the input for the shuffle element and same is forwarded by Shuffle to another task (or tasks) as input.

8.2.6 using NGSI Update to create it

Another way is to register a service topology by sending a constructed NGSI update message to the IoT Broker deployed in the cloud.

Here are the Curl and the Javascript-based code to register the service topology that is given in the above image. Users can take reference of the above service topology, i.e., anomaly detection to understand this code.

Note: In the Javascript code example, we use the Javascript-based library to interact with FogFlow IoT Broker. You can find out the library from the github code repository (designer/public/lib/ngsi). You must include ngsiclient.js into your web page.

Note: The Curl case assumes that the cloud IoT Broker is running on localhost on port 8070.

curl

JavaScript

```

curl -iX POST \
  'http://localhost:8070/ngsi10/updateContext' \
  -H 'Content-Type: application/json' \
  -d '
{
  "contextElements": [
    {
      "entityId":{
        "id":"Topology.anomaly-detection",
        "type":"Topology"
      },
      "attributes":[
        {
          "name":"status",
          "type":"string",
          "value":"enabled"
        },
        {
          "name":"designboard",
          "type":"object",
          "value":{
            "blocks":[
              {
                "id":1,
                "module":null,
                "type":"Task",
                "values":{
                  "name":"Counting",
                  "operator":"counter",
                  "outputs":[
                    "Stat"
                  ]
                }
              },
              {
                "x":202,
                "y":-146
              },
              {
                "id":2,
                "module":null,
                "type":"Task",
                "values":{
                  "name":"Detector",
                  "operator":"anomaly",
                  "outputs":[
                    "Anomaly"
                  ]
                }
              },
              {
                "x":-194,
                "y":-134
              },
              {
                "id":3,
                "module":null,

```

(continues on next page)

(continued from previous page)

```

        "type": "Shuffle",
        "values": {
            "groupby": "ALL",
            "selectedattributes": [
                "all"
            ]
        },
        "x": 4,
        "y": -18
    },
    {
        "id": 4,
        "module": null,
        "type": "EntityStream",
        "values": {
            "groupby": "EntityID",
            "scoped": true,
            "selectedattributes": [
                "all"
            ],
            "selectedtype": "PowerPanel"
        },
        "x": -447,
        "y": -179
    },
    {
        "id": 5,
        "module": null,
        "type": "EntityStream",
        "values": {
            "groupby": "ALL",
            "scoped": false,
            "selectedattributes": [
                "all"
            ],
            "selectedtype": "Rule"
        },
        "x": -438,
        "y": -5
    }
],
"edges": [
    {
        "block1": 3,
        "block2": 1,
        "connector1": [
            "stream",
            "output"
        ],
        "connector2": [
            "streams",
            "input"
        ]
    }
]

```

(continues on next page)

(continued from previous page)

```

    ],
    "id":2
  },
  {
    "block1":2,
    "block2":3,
    "connector1":[
      "outputs",
      "output",
      0
    ],
    "connector2":[
      "in",
      "input"
    ],
    "id":3
  },
  {
    "block1":4,
    "block2":2,
    "connector1":[
      "stream",
      "output"
    ],
    "connector2":[
      "streams",
      "input"
    ],
    "id":4
  },
  {
    "block1":5,
    "block2":2,
    "connector1":[
      "stream",
      "output"
    ],
    "connector2":[
      "streams",
      "input"
    ],
    "id":5
  }
]
},
{
  "name":"template",
  "type":"object",
  "value":{
    "description":"detect anomaly events in shops",
    "name":"anomaly-detection",

```

(continues on next page)

(continued from previous page)

```

      "tasks": [
        {
          "input_streams": [
            {
              "groupby": "ALL",
              "scoped": true,
              "selected_attributes": [
                ],
              "selected_type": "Anomaly"
            }
          ],
          "name": "Counting",
          "operator": "counter",
          "output_streams": [
            {
              "entity_type": "Stat"
            }
          ]
        },
        {
          "input_streams": [
            {
              "groupby": "EntityID",
              "scoped": true,
              "selected_attributes": [
                ],
              "selected_type": "PowerPanel"
            }
          ],
          {
            "groupby": "ALL",
            "scoped": false,
            "selected_attributes": [
              ],
            "selected_type": "Rule"
          }
        ],
        "name": "Detector",
        "operator": "anomaly",
        "output_streams": [
          {
            "entity_type": "Anomaly"
          }
        ]
      }
    ],
    "domainMetadata": [

```

(continues on next page)

(continued from previous page)

```

        {
            "name": "location",
            "type": "global",
            "value": "global"
        }
    ]
},
"updateAction": "UPDATE"
}'

```

// the json object that represent the structure of your service topology
// when using the FogFlow topology editor, this is generated by the editor

```

var topology = {
    "name": "template",
    "type": "object",
    "value": {
        "description": "detect anomaly events in shops",
        "name": "anomaly-detection",
        "tasks": [
            {
                "input_streams": [
                    {
                        "groupby": "ALL",
                        "scoped": true,
                        "selected_attributes": [

                        ],
                        "selected_type": "Anomaly"
                    }
                ],
                "name": "Counting",
                "operator": "counter",
                "output_streams": [
                    {
                        "entity_type": "Stat"
                    }
                ]
            },
            {
                "input_streams": [
                    {
                        "groupby": "EntityID",
                        "scoped": true,
                        "selected_attributes": [

                        ],
                        "selected_type": "PowerPanel"
                    }
                ],
                "groupby": "ALL",
                "scoped": false,

```

(continues on next page)

(continued from previous page)

```

        "selected_attributes": [
            ],
            "selected_type": "Rule"
        }
    ],
    "name": "Detector",
    "operator": "anomaly",
    "output_streams": [
        {
            "entity_type": "Anomaly"
        }
    ]
}
}

var design = {
    "name": "designboard",
    "type": "object",
    "value": {
        "blocks": [
            {
                "id": 1,
                "module": null,
                "type": "Task",
                "values": {
                    "name": "Counting",
                    "operator": "counter",
                    "outputs": [
                        "Stat"
                    ]
                }
            },
            {
                "x": 202,
                "y": -146
            },
            {
                "id": 2,
                "module": null,
                "type": "Task",
                "values": {
                    "name": "Detector",
                    "operator": "anomaly",
                    "outputs": [
                        "Anomaly"
                    ]
                }
            },
            {
                "x": -194,
                "y": -134
            },
            {

```

(continues on next page)

(continued from previous page)

```

        "id":3,
        "module":null,
        "type":"Shuffle",
        "values":{
            "groupby":"ALL",
            "selectedattributes":[
                "all"
            ]
        },
        "x":4,
        "y":-18
    },
    {
        "id":4,
        "module":null,
        "type":"EntityStream",
        "values":{
            "groupby":"EntityID",
            "scoped":true,
            "selectedattributes":[
                "all"
            ],
            "selectedtype":"PowerPanel"
        },
        "x":-447,
        "y":-179
    },
    {
        "id":5,
        "module":null,
        "type":"EntityStream",
        "values":{
            "groupby":"ALL",
            "scoped":false,
            "selectedattributes":[
                "all"
            ],
            "selectedtype":"Rule"
        },
        "x":-438,
        "y":-5
    }
],
"edges":[
    {
        "block1":3,
        "block2":1,
        "connector1":[
            "stream",
            "output"
        ],
        "connector2":[]
    }
]

```

(continues on next page)

(continued from previous page)

```

        "streams",
        "input"
    ],
    "id":2
},
{
    "block1":2,
    "block2":3,
    "connector1":[
        "outputs",
        "output",
        0
    ],
    "connector2":[
        "in",
        "input"
    ],
    "id":3
},
{
    "block1":4,
    "block2":2,
    "connector1":[
        "stream",
        "output"
    ],
    "connector2":[
        "streams",
        "input"
    ],
    "id":4
},
{
    "block1":5,
    "block2":2,
    "connector1":[
        "stream",
        "output"
    ],
    "connector2":[
        "streams",
        "input"
    ],
    "id":5
}
]
}

```

```

//submit it to FogFlow via NGSI Update
var topologyCtxObj = {};

```

(continues on next page)

(continued from previous page)

```

topologyCtxObj.entityId = {
  id : 'Topology.' + topology.value.name,
  type: 'Topology',
  isPattern: false
};

topologyCtxObj.attributes = {};
topologyCtxObj.attributes.status = {type: 'string', value: 'enabled'};
topologyCtxObj.attributes.designboard = design;
topologyCtxObj.attributes.template = topology;

// assume the config.brokerURL is the IP of cloud IoT Broker
var client = new NGSI10Client(config.brokerURL);

// send NGSI10 update
client.updateContext(topologyCtxObj).then( function(data) {
  console.log(data);
}).catch( function(error) {
  console.log('failed to submit the topology');
});

```

8.2.7 Trigger the service topology by sending an Intent

Once developers submit a specified service topology and the implemented operators, the service data processing logic can be triggered by following two steps:

- Sending a high level intent object which breaks the service topology into separate tasks
- Providing Input Streams to the tasks of that service topology.

The intent object is sent using the fogflow dashboard with the following properties:

- Topology: specifies which topology the intent object is meant for.
- Priority: defines the priority level of all tasks in your topology, which will be utilized by edge nodes to decide how resources should be assigned to the tasks.
- Resource Usage: defines how a topology can use resources on edge nodes. Sharing in an exclusive way means the topology will not share the resources with any task from other topologies. The other way is inclusive one.
- Objective: of maximum throughput, minimum latency and minimum cost can be set for task assignment at workers. However, this feature is not fully supported yet, so it can be set as “None” for now.
- Geoscope: is a defined geographical area where input streams should be selected. Global as well as custom geoscopes can be set.

Fogflow topology master will now be waiting for input streams for the tasks contained in the service topology. As soon as context data are received, which fall within the scope of the intent object, tasks are launched on the nearest workers.

Here are curl examples to send Input streams for Anomaly-Detector use case. It requires PowerPanel as well as Rule data.

Note: Users can also use to send PowerPanel data.

FogFlow
System Status
Operator Registry
Service Topology
Fog Function
NGSI-v1 Apps
NGSI-LD Apps

Service Topology
Service Intent
Task Instance

to specify an intent object in order to run your service

Topology
anomaly-detection
Type
Synchronous
Priority
low
Resource usage
inclusive
Objective
None
Geoscope
global

Apply

Note: The Curl case assumes that the cloud IoT Broker is running on localhost on port 8070.

```
curl -iX POST \
  'http://localhost:8070/ngsi10/updateContext' \
-H 'Content-Type: application/json' \
-d '
{
  "contextElements": [
    {
      "entityId":{
        "id":"Device.PowerPanel.01",
        "type":"PowerPanel"
      },
      "attributes":[
        {
          "name":"usage",
          "type":"integer",
          "value":4
        },
        {
          "name":"shop",
          "type":"string",
          "value":"01"
        },
        {
          "name":"iconURL",
          "type":"string",
          "value":"/img/shop.png"
        }
      ]
    }
  ]
}
```

(continues on next page)

(continued from previous page)

```

    }
  ],
  "domainMetadata": [
    {
      "name": "location",
      "type": "point",
      "value": {
        "latitude": 35.7,
        "longitude": 138
      }
    },
    {
      "name": "shop",
      "type": "string",
      "value": "01"
    }
  ]
} ],
"updateAction": "UPDATE"
}'

```

```

curl -iX POST \
  'http://localhost:8070/ngsi10/updateContext' \
-H 'Content-Type: application/json' \
-d '
{
  "contextElements": [
    {
      "entityId": {
        "id": "Stream.Rule.01",
        "type": "Rule"
      },
      "attributes": [
        {
          "name": "threshold",
          "type": "integer",
          "value": 30
        }
      ]
    }
  ],
  "updateAction": "UPDATE"
}'

```


API WALKTHROUGH

9.1 FogFlow Discovery API

9.1.1 Look up nearby brokers

For any external application or IoT devices, the only interface they need from FogFlow Discovery is to find out a nearby Broker based on its own location. After that, they only need to interact with the assigned nearby Broker.

POST /ngsi9/discoverContextAvailability

Param	Description
latitude	latitude of your location
longitude	latitude of your location
limit	number of expected brokers

Please check the following examples.

Note: For the Javascript code example, library `ngsiclient.js` is needed. Please refer to the code repository at [application/device/powerpanel](#)

curl

JavaScript

```
curl -iX POST \
  'http://localhost:80/ngsi9/discoverContextAvailability' \
  -H 'Content-Type: application/json' \
  -d '
  {
    "entities":[
      {
        "type":"IoTBroker",
        "isPattern":true
      }
    ],
    "restriction":{
      "scopes":[
        {
          "scopeType":"nearby",
          "scopeValue":{
```

(continues on next page)

(continued from previous page)

```
        "latitude":35.692221,  
        "longitude":139.709059,  
        "limit":1  
      }  
    }  
  ]  
}  
'
```

```
const NGSI = require('./ngsi/ngsiclient.js');  
  
var discoveryURL = "http://localhost:80/ngsi9";  
var myLocation = {  
  "latitude": 35.692221,  
  "longitude": 139.709059  
};  
  
// find out the nearby IoT Broker according to my location  
var discovery = new NGSI.NGSI9Client(discoveryURL)  
discovery.findNearbyIoTBroker(myLocation, 1).then( function(brokers) {  
  console.log('-----nearbybroker-----');  
  console.log(brokers);  
  console.log('-----end-----');  
}).catch(function(error) {  
  console.log(error);  
});
```

9.2 FogFlow Broker API

Note: Use port 80 for accessing the cloud broker, whereas for edge broker, the default port is 8070.

9.2.1 Create/update context

Note: It is the same API to create or update a context entity. For a context update, if there is no existing entity, a new entity will be created.

POST /ngsi10/updateContext

Param	Description
latitude	latitude of your location
longitude	latitude of your location
limit	number of expected brokers

Example:

curl

JavaScript

```
curl -iX POST \
  'http://localhost:80/ngsi10/updateContext' \
  -H 'Content-Type: application/json' \
  -d '
  {
    "contextElements": [
      {
        "entityId": {
          "id": "Device.temp001",
          "type": "Temperature",
          "isPattern": false
        },
        "attributes": [
          {
            "name": "temp",
            "type": "integer",
            "value": 10
          }
        ],
        "domainMetadata": [
          {
            "name": "location",
            "type": "point",
            "value": {
              "latitude": 49.406393,
              "longitude": 8.684208
            }
          }, {
            "name": "city",
            "type": "string",
            "value": "Heidelberg"
          }
        ]
      }
    ],
    "updateAction": "UPDATE"
  }'
```

```
const NGSI = require('./ngsi/ngsiclient.js');
var brokerURL = "http://localhost:80/ngsi10"

var ngsi10client = new NGSI.NGSI10Client(brokerURL);

var profile = {
  "type": "PowerPanel",
  "id": "01"};

var ctxObj = {};
ctxObj.entityId = {
  id: 'Device.' + profile.type + '.' + profile.id,
```

(continues on next page)

(continued from previous page)

```

    type: profile.type,
    isPattern: false
  };

  ctxObj.attributes = {};

  var degree = Math.floor((Math.random() * 100) + 1);
  ctxObj.attributes.usage = {
    type: 'integer',
    value: degree
  };
  ctxObj.attributes.shop = {
    type: 'string',
    value: profile.id
  };
  ctxObj.attributes.iconURL = {
    type: 'string',
    value: profile.iconURL
  };

  ctxObj.metadata = {};

  ctxObj.metadata.location = {
    type: 'point',
    value: profile.location
  };

  ngsi10client.updateContext(ctxObj).then( function(data) {
    console.log(data);
  }).catch(function(error) {
    console.log('failed to update context');
  });

```

9.2.2 Query Context via GET

Fetch a context entity by ID

GET /ngsi10/entity/#eid

Param	Description
eid	entity ID

Example:

```
curl http://localhost:80/ngsi10/entity/Device.temp001
```

Fetch a specific attribute of a specific context entity

GET /ngsi10/entity/#eid/#attr

Param	Description
eid	entity ID
attr	specify the attribute name to be fetched

Example:

```
curl http://localhost:80/ngsi10/entity/Device.temp001/temp
```

Check all context entities on a single Broker

GET /ngsi10/entity

Example:

```
curl http://localhost:80/ngsi10/entity
```

9.2.3 Query context via POST

POST /ngsi10/queryContext

Param	Description
entityId	specify the entity filter, which can define a specific entity ID, ID pattern, or type
restriction	a list of scopes and each scope defines a filter based on domain metadata

query context by the pattern of entity ID

curl

JavaScript

```
curl -X POST 'http://localhost:80/ngsi10/queryContext' \
-H 'Content-Type: application/json' \
-d '{"entities":[{"id":"Device.*","isPattern":true}]}'
```

```
const NGSI = require('./ngsi/ngsiclient.js');
var brokerURL = "http://localhost:80/ngsi10"
var ngsi10client = new NGSI.NGSI10Client(brokerURL);

var queryReq = {}
queryReq.entities = [{id:'Device.*', isPattern: true}];

ngsi10client.queryContext(queryReq).then( function(deviceList) {
    console.log(deviceList);
}).catch(function(error) {
    console.log(error);
    console.log('failed to query context');
});
```

query context by entity type

curl

JavaScript

```
curl -X POST 'http://localhost:80/ngsi10/queryContext' \  
  -H 'Content-Type: application/json' \  
  -d '{"entities":[{"type":"Temperature","isPattern":true}]}'
```

```
const NGSI = require('./ngsi/ngsiclient.js');  
var brokerURL = "http://localhost:80/ngsi10"  
var ngsi10client = new NGSI.NGSI10Client(brokerURL);  
  
var queryReq = {}  
queryReq.entities = [{type:'Temperature', isPattern: true}];  
  
ngsi10client.queryContext(queryReq).then( function(deviceList) {  
  console.log(deviceList);  
}).catch(function(error) {  
  console.log(error);  
  console.log('failed to query context');  
});
```

query context by geo-scope (circle)

curl

JavaScript

```
curl -X POST 'http://localhost:80/ngsi10/queryContext' \  
  -H 'Content-Type: application/json' \  
  -d '{  
    "entities": [{  
      "id": ".*",  
      "isPattern": true  
    }],  
    "restriction": {  
      "scopes": [{  
        "scopeType": "circle",  
        "scopeValue": {  
          "centerLatitude": 49.406393,  
          "centerLongitude": 8.684208,  
          "radius": 10.0  
        }  
      }]  
    }  
  }'
```

```
const NGSI = require('./ngsi/ngsiclient.js');  
var brokerURL = "http://localhost:80/ngsi10"  
var ngsi10client = new NGSI.NGSI10Client(brokerURL);
```

(continues on next page)

(continued from previous page)

```

var queryReq = {}
queryReq.entities = [{type: '.*', isPattern: true}];
queryReq.restriction = {scopes: [{
    "scopeType": "circle",
    "scopeValue": {
        "centerLatitude": 49.406393,
        "centerLongitude": 8.684208,
        "radius": 10.0
    }
}]}];

ngsi10client.queryContext(queryReq).then( function(deviceList) {
    console.log(deviceList);
}).catch(function(error) {
    console.log(error);
    console.log('failed to query context');
});

```

query context by geo-scope (polygon)

curl

JavaScript

```

curl -X POST 'http://localhost:80/ngsi10/queryContext' \
-H 'Content-Type: application/json' \
-d '{
  "entities": [
    {
      "id": ".*",
      "isPattern": true
    }
  ],
  "restriction": {
    "scopes": [
      {
        "scopeType": "polygon",
        "scopeValue": {
          "vertices": [
            {
              "latitude": 34.4069096565206,
              "longitude": 135.84594726562503
            },
            {
              "latitude": 37.18657859524883,
              "longitude": 135.84594726562503
            },
            {
              "latitude": 37.18657859524883,
              "longitude": 141.51489257812503
            }
          ]
        }
      }
    ]
  }
}';

```

(continues on next page)

(continued from previous page)

```

        {
          "latitude":34.4069096565206,
          "longitude":141.51489257812503
        },
        {
          "latitude":34.4069096565206,
          "longitude":135.84594726562503
        }
      ]
    }
  }
}

```

```

const NGSI = require('./ngsi/ngsiclient.js');
var brokerURL = "http://localhost:80/ngsi10"
var ngsi10client = new NGSI.NGSI10Client(brokerURL);

var queryReq = {}
queryReq.entities = [{type:'.*', isPattern: true}];
queryReq.restriction = {
  "scopes":[
    {
      "scopeType":"polygon",
      "scopeValue":{
        "vertices":[
          {
            "latitude":34.4069096565206,
            "longitude":135.84594726562503
          },
          {
            "latitude":37.18657859524883,
            "longitude":135.84594726562503
          },
          {
            "latitude":37.18657859524883,
            "longitude":141.51489257812503
          },
          {
            "latitude":34.4069096565206,
            "longitude":141.51489257812503
          },
          {
            "latitude":34.4069096565206,
            "longitude":135.84594726562503
          }
        ]
      }
    }
  ]
}

```

(continues on next page)

(continued from previous page)

```
ngsi10client.queryContext(queryReq).then( function(deviceList) {
    console.log(deviceList);
}).catch(function(error) {
    console.log(error);
    console.log('failed to query context');
});
```

query context with the filter of domain metadata values

Note: the conditional statement can be defined only with the domain metadata of your context entities For the time being, it is not supported to filter out entities based on specific attribute values.

curl

JavaScript

```
curl -X POST 'http://localhost:80/ngsi10/queryContext' \
-H 'Content-Type: application/json' \
-d '{
  "entities": [{
    "id": ".*",
    "isPattern": true
  }],
  "restriction": {
    "scopes": [{
      "scopeType": "stringQuery",
      "scopeValue": "city=Heidelberg"
    }]
  }
}'
```

```
const NGSI = require('./ngsi/ngsiclient.js');
var brokerURL = "http://localhost:80/ngsi10"
var ngsi10client = new NGSI.NGSI10Client(brokerURL);

var queryReq = {}
queryReq.entities = [{type: '.*', isPattern: true}];
queryReq.restriction = {scopes: [{
  "scopeType": "stringQuery",
  "scopeValue": "city=Heidelberg"
}]}];

ngsi10client.queryContext(queryReq).then( function(deviceList) {
    console.log(deviceList);
}).catch(function(error) {
    console.log(error);
    console.log('failed to query context');
});
```

query context with multiple filters

curl

JavaScript

```
curl -X POST 'http://localhost:80/ngsi10/queryContext' \  
-H 'Content-Type: application/json' \  
-d '{  
  "entities": [{  
    "id": ".*",  
    "isPattern": true  
  }],  
  "restriction": {  
    "scopes": [{  
      "scopeType": "circle",  
      "scopeValue": {  
        "centerLatitude": 49.406393,  
        "centerLongitude": 8.684208,  
        "radius": 10.0  
      }  
    }, {  
      "scopeType": "stringQuery",  
      "scopeValue": "city=Heidelberg"  
    }  
  ]  
}'
```

```
const NGSI = require('./ngsi/ngsiclient.js');  
var brokerURL = "http://localhost:80/ngsi10"  
var ngsi10client = new NGSI.NGSI10Client(brokerURL);  
  
var queryReq = {}  
queryReq.entities = [{type: '.*', isPattern: true}];  
queryReq.restriction = {scopes: [{  
  "scopeType": "circle",  
  "scopeValue": {  
    "centerLatitude": 49.406393,  
    "centerLongitude": 8.684208,  
    "radius": 10.0  
  }  
}, {  
  "scopeType": "stringQuery",  
  "scopeValue": "city=Heidelberg"  
}]}];  
  
ngsi10client.queryContext(queryReq).then( function(deviceList) {  
  console.log(deviceList);  
}).catch(function(error) {  
  console.log(error);  
  console.log('failed to query context');  
});
```

9.2.4 Delete context

Delete a specific context entity by ID

DELETE /ngsi10/entity/#eid

Param	Description
eid	entity ID

Example:

```
curl -iX DELETE http://localhost:80/ngsi10/entity/Device.temp001
```

9.2.5 Subscribe context

POST /ngsi10/subscribeContext

Param	Description
entityId	specify the entity filter, which can define a specific entity ID, ID pattern, or type
restriction	a list of scopes and each scope defines a filter based on domain metadata
reference	the destination to receive notifications

subscribe context by the pattern of entity ID

curl

JavaScript

```
curl -X POST 'http://localhost:80/ngsi10/subscribeContext' \
-H 'Content-Type: application/json' \
-d '{
  "entities":[{"id":"Device.*","isPattern":true}],
  "reference": "http://localhost:8066"
}'
```

```
const NGSI = require('./ngsi/ngsiclient.js');
var brokerURL = "http://localhost:80/ngsi10"
var ngsi10client = new NGSI.NGSI10Client(brokerURL);
var mySubscriptionId;

var subscribeReq = {}
subscribeReq.entities = [{id:'Device.*', isPattern: true}];

ngsi10client.subscribeContext(subscribeReq).then( function(subscriptionId) {
  console.log("subscription id = " + subscriptionId);
  mySubscriptionId = subscriptionId;
}).catch(function(error) {
  console.log('failed to subscribe context');
});
```

subscribe context by entity type

curl

JavaScript

```
curl -X POST 'http://localhost:80/ngsi10/subscribeContext' \
-H 'Content-Type: application/json' \
-d '{
  "entities": [{"type": "Temperature", "isPattern": true}]
  "reference": "http://localhost:8066"
}'
```

```
const NGSI = require('./ngsi/ngsiclient.js');
var brokerURL = "http://localhost:80/ngsi10"
var ngsi10client = new NGSI.NGSI10Client(brokerURL);

var subscribeReq = {}
subscribeReq.entities = [{type: 'Temperature', isPattern: true}];

ngsi10client.subscribeContext(subscribeReq).then( function(subscriptionId) {
  console.log("subscription id = " + subscriptionId);
  mySubscriptionId = subscriptionId;
}).catch(function(error) {
  console.log('failed to subscribe context');
});
```

subscribe context by geo-scope

curl

JavaScript

```
curl -X POST 'http://localhost:80/ngsi10/subscribeContext' \
-H 'Content-Type: application/json' \
-d '{
  "entities": [{
    "id": ".*",
    "isPattern": true
  }],
  "reference": "http://localhost:8066",
  "restriction": {
    "scopes": [{
      "scopeType": "circle",
      "scopeValue": {
        "centerLatitude": 49.406393,
        "centerLongitude": 8.684208,
        "radius": 10.0
      }
    }]
  }
}'
```

```

const NGSI = require('./ngsi/ngsiclient.js');
var brokerURL = "http://localhost:80/ngsi10"
var ngsi10client = new NGSI.NGSI10Client(brokerURL);

var subscribeReq = {}
subscribeReq.entities = [{type: '.*', isPattern: true}];
subscribeReq.restriction = {scopes: [{
    "scopeType": "circle",
    "scopeValue": {
        "centerLatitude": 49.406393,
        "centerLongitude": 8.684208,
        "radius": 10.0
    }
}
]
}
];

ngsi10client.subscribeContext(subscribeReq).then( function(subscriptionId) {
    console.log("subscription id = " + subscriptionId);
    mySubscriptionId = subscriptionId;
}).catch(function(error) {
    console.log('failed to subscribe context');
});

```

subscribe context with the filter of domain metadata values

Note: the conditional statement can be defined only with the domain metadata of your context entities. For the time being, it is not supported to filter out entities based on specific attribute values.

curl

JavaScript

```

curl -X POST 'http://localhost:80/ngsi10/subscribeContext' \
-H 'Content-Type: application/json' \
-d '{
    "entities": [{
        "id": ".*",
        "isPattern": true
    }],
    "reference": "http://localhost:8066",
    "restriction": {
        "scopes": [{
            "scopeType": "stringQuery",
            "scopeValue": "city=Heidelberg"
        }
    ]
}
}'

```

```

const NGSI = require('./ngsi/ngsiclient.js');
var brokerURL = "http://localhost:80/ngsi10"
var ngsi10client = new NGSI.NGSI10Client(brokerURL);

```

(continues on next page)

(continued from previous page)

```

var subscribeReq = {}
subscribeReq.entities = [{type: '.*', isPattern: true}];
subscribeReq.restriction = {scopes: [{
    "scopeType": "stringQuery",
    "scopeValue": "city=Heidelberg"
}]};

ngsi10client.subscribeContext(subscribeReq).then( function(subscriptionId) {
    console.log("subscription id = " + subscriptionId);
    mySubscriptionId = subscriptionId;
}).catch(function(error) {
    console.log('failed to subscribe context');
});

```

subscribe context with multiple filters

curl

JavaScript

```

curl -X POST 'http://localhost:80/ngsi10/subscribeContext' \
-H 'Content-Type: application/json' \
-d '{
    "entities": [{
        "id": ".*",
        "isPattern": true
    }],
    "reference": "http://localhost:8066",
    "restriction": {
        "scopes": [{
            "scopeType": "circle",
            "scopeValue": {
                "centerLatitude": 49.406393,
                "centerLongitude": 8.684208,
                "radius": 10.0
            }
        }, {
            "scopeType": "stringQuery",
            "scopeValue": "city=Heidelberg"
        }]
    }
}'

```

```

const NGSI = require('./ngsi/ngsiclient.js');
var brokerURL = "http://localhost:80/ngsi10"
var ngsi10client = new NGSI.NGSI10Client(brokerURL);

var subscribeReq = {}
subscribeReq.entities = [{type: '.*', isPattern: true}];
subscribeReq.restriction = {scopes: [{

```

(continues on next page)

(continued from previous page)

```

        "scopeType": "circle",
        "scopeValue": {
            "centerLatitude": 49.406393,
            "centerLongitude": 8.684208,
            "radius": 10.0
        }
    }, {
        "scopeType": "stringQuery",
        "scopeValue": "city=Heidelberg"
    }
  ]
};

// use the IP and Port number your receiver is listening
subscribeReq.reference = 'http://' + agentIP + ':' + agentPort;

ngsi10client.subscribeContext(subscribeReq).then( function(subscriptionId) {
    console.log("subscription id = " + subscriptionId);
    mySubscriptionId = subscriptionId;
}).catch(function(error) {
    console.log('failed to subscribe context');
});

```

Cancel a subscription by subscription ID

DELETE /ngsi10/subscription/#sid

Param	Description
sid	the subscription ID created when the subscription is issued

curl -iX DELETE http://localhost:80/ngsi10/subscription/#sid

9.3 FogFlow Designer API

FogFlow uses its own REST APIs to manage all internal objects, including operator, docker image, service topology, service intent, and fog function. In addition, FogFlow also provides the extra interface for device registration and the management of subscriptions to exchange data with other FIWARE brokers, such as Orion/Orion-LD and Scorpio, which could be used both as the data sources to fetch the original data or as the destination to publish the generated results.

9.3.1 Operator

a. To create a new Operator

POST /operator

Example

```
curl -X POST \
  'http://127.0.0.1:8080/operator' \
  -H 'Content-Type: application/json' \
  -d '
    [{
      "name": "dummy",
      "description": "test",
      "parameters": []
    }]
  '
```

b. To retrieve all the operators

GET /operator

Example:

```
curl -iX GET \
  'http://127.0.0.1:8080/operator'
```

c. To retrieve a specific operator based on operator name

GET /operator/<name>

Param	Description
name	Name of existing operator

Example:

```
curl -iX GET \
  'http://127.0.0.1:8080/operator/dummy'
```

9.3.2 DockerImage

a. To create a new DockerImage

POST /dockerimage

Example

```
curl -X POST \
  'http://127.0.0.1:8080/dockerimage' \
  -H 'Content-Type: application/json' \
  -d '
    [
      {
        "name": "fogflow/dummy",
        "hwType": "X86",
        "osType": "Linux",
        "operatorName": "dummy",

```

(continues on next page)

(continued from previous page)

```

        "prefetched": false,
        "tag": "latest"
    }
]
,
```

b. To retrieve all the DockerImage**GET /dockerimage****Example:**

```
curl -iX GET \
'http://127.0.0.1:8080/dockerimage'
```

c. To retrieve a specific DockerImage based on operator name**GET /dockerimage/<operator name>**

Param	Description
name	Name of existing operator

Example:

```
curl -iX GET \
'http://127.0.0.1:8080/dockerimage/dummy'
```

9.3.3 Service Topology**a. To create a new Service****POST /service****Example**

```
curl -X POST \
'http://127.0.0.1:8080/service' \
-H 'Content-Type: application/json' \
-d '
[
  {
    "topology": {
      "name": "MyTest",
      "description": "a simple case",
      "tasks": [
        {
          "name": "main",
          "operator": "dummy",
          "input_streams": [
```

(continues on next page)

(continued from previous page)

```

        {
            "selected_type": "Temperature",
            "selected_attributes": [],
            "groupby": "EntityID",
            "scoped": false
        },
        "output_streams": [
            {
                "entity_type": "Out"
            }
        ]
    }
]

```

b. To retrieve all the Service**GET /service****Example:**

```
curl -iX GET \
'http://127.0.0.1:8080/service'
```

c. To retrieve a specific service based on service name**GET /service/<service name>**

Param	Description
name	Name of existing service

Example:

```
curl -iX GET \
'http://127.0.0.1:8080/service/MyTest'
```

d. To delete a specific service based on service name**DELETE /service/<service name>**

Param	Description
name	Name of existing service

Example:

```
curl -X DELETE 'http://localhost:8080/service/MyTest' \
-H 'Content-Type: application/json'
```

9.3.4 Intent

a. To create a new Intent

POST /intent

Example

```
curl -X POST \
'http://127.0.0.1:8080/intent' \
-H 'Content-Type: application/json' \
-d '
{
  "id": "ServiceIntent.594e3d10-59f9-4ee6-97be-fe50b9c99bd8",
  "topology": "MyTest",
  "stype": "ASYN",
  "priority": {
    "exclusive": false,
    "level": 0
  },
  "qos": "NONE",
  "geoscope": {
    "scopeType": "global",
    "scopeValue": "global"
  }
}
```

b. To retrieve all the Intent

GET /intent

Example:

```
curl -iX GET \
'http://127.0.0.1:8080/intent'
```

c. To retrieve a specific Intent based on Intent ID

GET /intent/<intent id>

Param	Description
id	ID of existing intent

Example:

```
curl -iX GET \  
  'http://127.0.0.1:8080/intent/ServiceIntent.594e3d10-59f9-4ee6-97be-fe50b9c99bd8'
```

d. To delete a specific Intent based on intent id

DELETE /intent/<intent id>

Param	Description
id	ID of the existing intent

Example:

```
curl -iX DELETE \  
  'http://127.0.0.1:8080/intent/ServiceIntent.594e3d10-59f9-4ee6-97be-fe50b9c99bd8'
```

e. To retrieve list service intents for the given service topology

GET /intent/topology/<TopologeName>

Param	Description
TopologeName	name of the given service topology

Example:

```
curl -iX GET \  
  'http://127.0.0.1:8080/intent/topology/MyTest'
```

9.3.5 Topology

a. To retrieve all the Topology

GET /topology

Example:

```
curl -X GET 'http://localhost:8080/topology' \  
-H 'Content-Type: application/json'
```

b. To retrieve a specific Topology based on topology name**GET /topology/<topology name>**

Param	Description
Name	name of the existing Topology

Example:

```
curl -X GET 'http://localhost:8080/topology/MyTest' \
-H 'Content-Type: application/json'
```

9.3.6 Fog Function**a. To create a new Fogfunction****POST /fogfunction****Example**

```
curl -X POST \
'http://127.0.0.1:8080/fogfunction' \
-H 'Content-Type: application/json' \
-d '
    [
      {
        "name": "ffTest",
        "topology": {
          "name": "ffTest",
          "description": "a fog function",
          "tasks": [
            {
              "name": "main",
              "operator": "mqtt-adapter",
              "input_streams": [
                {
                  "selected_type": "HOPU",
                  "selected_attributes": [],
                  "groupby": "EntityID",
                  "scoped": false
                }
              ],
              "output_streams": [
                {
                  "entity_type": "Out"
                }
              ]
            }
          ]
        },
        "intent": {
          "id": "ServiceIntent.1c6396bb-281d-4c14-b61d-f0cc0dcc1006",
```

(continues on next page)

(continued from previous page)

```
        "topology": "ffTest",
        "priority": {
          "exclusive": false,
          "level": 0
        },
        "qos": "default",
        "geoscope": {
          "scopeType": "global",
          "scopeValue": "global"
        },
        "status": "enabled"
      },
    ]
  },
}
```

b. To retrieve all the Fogfunction

GET /fogfunction

Example:

```
curl -iX GET \
'http://127.0.0.1:8080/fogfunction'
```

c. To retrieve a specific Fogfunction based on fogfunction name

GET /fogfunction/<name>

Param	Description
name	Name of existing fogfunction

Example:

```
curl -iX GET \
'http://127.0.0.1:8080/fogfunction/ffTest'
```

d. To delete a specific fogfunction based on fogfunction name

DELETE /fogfunction/<fogfunction name>

Param	Description
name	Name of existing fogfunction

Example:

```
curl -iX DELETE \
'http://127.0.0.1:8080/fogfunction/ffTest'
```

e. To enable a specific fogfunction based on fogfunction name**GET /fogfunction/<fogfunction name>/enable**

Param	Description
name	Name of existing fogfunction

Example:

```
curl -iX GET \
  'http://127.0.0.1:8080/fogfunction/ffTest/enable'
```

F. To disable a specific fogfunction based on fogfunction name**GET /fogfunction/<fogfunction name>/disable**

Param	Description
name	Name of existing fogfunction

Example:

```
curl -iX GET \
  'http://127.0.0.1:8080/fogfunction/ffTest/disable'
```

9.3.7 Device**a. To create a new device****POST /device****Example**

```
curl -iX POST \
  'http://127.0.0.1:8080/device' \
  -H 'Content-Type: application/json' \
  -d '
  {
    "id": "urn:Device.12345",
    "type": "HOPU",
    "attributes": {
      "protocol": {
        "type": "string",
        "value": "MQTT"
      },
      "mqttbroker": {
        "type": "string",
        "value": "mqtt://mqtt.cdtidev.nec-ccoc.com:1883"
      },
      "topic": {
        "type": "string",
```

(continues on next page)

(continued from previous page)

```

        "value": "/api/12345/attrs"
      },
      "mappings": {
        "type": "object",
        "value": {
          "temp8": {
            "name": "temperature",
            "type": "float",
            "entity_type": "AirQualityObserved"
          },
          "hum8": {
            "name": "humidity",
            "type": "float",
            "entity_type": "AirQualityObserved"
          }
        }
      }
    },
    "metadata": {
      "location": {
        "type": "point",
        "value": {
          "latitude": 49.406393,
          "longitude": 8.684208
        }
      }
    }
  }
}'

```

b. To get the list of all registered devices

GET /device

Example:

```
curl -iX GET \
  'http://127.0.0.1:8080/device'
```

c. To delete a specific device

DELETE /device/<device_id>

Param	Description
device_id	entity ID of this device

Example:

```
curl -iX DELETE \
  'http://127.0.0.1:8080/device/urn:Device.12345'
```


9.3.8 Subscription

a. To create a subscription for a given destination

POST /subscription

Example

```
curl -iX POST \
  'http://127.0.0.1:8080/subscription' \
  -H 'Content-Type: application/json' \
  -d '{
    "entity_type": "AirQualityObserved",
    "destination_broker": "NGSI-LD",
    "reference_url": "http://127.0.0.1:9090",
    "tenant": "ccoc"
  }'
```

b. To get the list of all registered subscription

GET /subscription

Example:

```
curl -iX GET \
  'http://127.0.0.1:8080/subscription'
```

c. To delete a specific subscription

DELETE /subscription/<subscription_id>

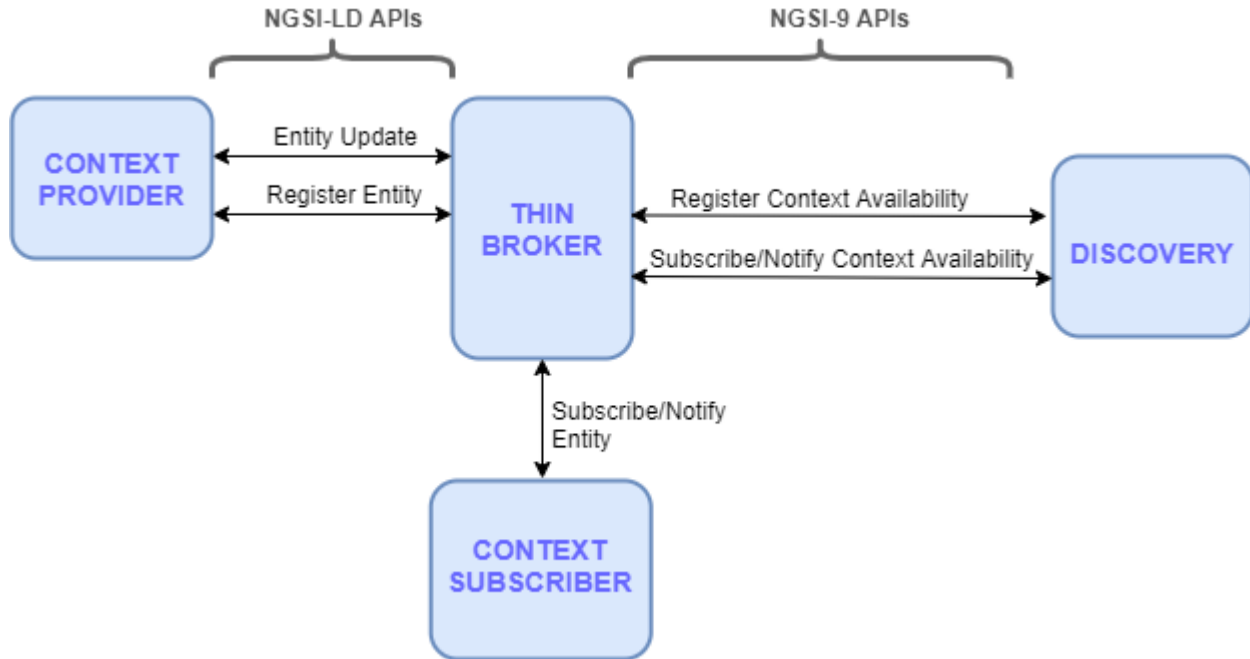
Param	Description
subscription_id	ID of this subscription

Example:

```
curl -iX DELETE \
  'http://127.0.0.1:8080/subscription/88bba05c-dda2-11ec-ba1d-acde48001122'
```

9.4 NGSI-LD Supported API's

The following figure shows a brief overview of how the APIs in current scope will be used to achieve the goal of NGSI-LD API support in FogFlow. The API support includes Entity creation, registration, subscription and notification.



9.4.1 Entities API

For the purpose of interaction with Fogflow, IOT devices approaches broker with entity creation request where it is resolved as per given context. Broker further forwards the registration request to Fogflow Discovery in correspondence to the created entity.

Note: Use port 80 for accessing the cloud broker, whereas for edge broker, the default port is 8070. The localhost is the coreservice IP for the system hosting fogflow.

POST /ngsi-ld/v1/entities/

a. To create NGSI-LD context entity, with context in Link in Header

key	Value
Content-Type	application/json
Accept	application/ld+json
Link	<{{link}}>; rel="https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"; type="application/ld+json"

Request

```

curl -iX POST \
  'http://localhost:80/ngsi-ld/v1/entities/' \
  -H 'Content-Type: application/json' \
  -H 'Accept: application/ld+json' \
  -H 'Link: <{{link}}>; rel="https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"; type="application/ld+json"' \
  <{{body}}
  
```

(continues on next page)

(continued from previous page)

```
-d '{
  "id": "urn:ngsi-ld:Vehicle:A100",
  "type": "Vehicle",
  "brandName": {
    "type": "Property",
    "value": "Mercedes"
  },
  "isParked": {
    "type": "Relationship",
    "object": "urn:ngsi-ld:OffStreetParking:Downtown1",
    "observedAt": "2017-07-29T12:00:04",
    "providedBy": {
      "type": "Relationship",
      "object": "urn:ngsi-ld:Person:Bob"
    }
  },
  "speed": {
    "type": "Property",
    "value": 80
  },
  "createdAt": "2017-07-29T12:00:04",
  "location": {
    "type": "GeoProperty",
    "value": {
      "type": "Point",
      "coordinates": [-8.5, 41.2]
    }
  }
}'
```

b. To create a new NGSI-LD context entity, with context in Link header and request payload is already expanded

key	Value
Content-Type	application/json
Accept	application/ld+json

Request

```
curl -iX POST \
'http://localhost:80/ngsi-ld/v1/entities/' \
-H 'Content-Type: application/json' \
-H 'Accept: application/ld+json' \
-d'
{
  "http://example.org/vehicle/brandName": [
    {
      "@type": [
        "http://uri.etsi.org/ngsi-ld/Property"
```

(continues on next page)

(continued from previous page)

```

        ],
        "http://uri.etsi.org/ngsi-ld/hasValue": [
            {
                "@value": "Mercedes"
            }
        ]
    },
    ],
    "http://uri.etsi.org/ngsi-ld/createdAt": [
        {
            "@type": "http://uri.etsi.org/ngsi-ld/DateTime",
            "@value": "2017-07-29T12:00:04"
        }
    ],
    "id": "urn:ngsi-ld:Vehicle:A8866",
    "http://example.org/common/isParked": [
        {
            "http://uri.etsi.org/ngsi-ld/hasObject": [
                {
                    "@id": "urn:ngsi-ld:OffStreetParking:Downtown1"
                }
            ],
            "http://uri.etsi.org/ngsi-ld/observedAt": [
                {
                    "@type": "http://uri.etsi.org/ngsi-ld/DateTime",
                    "@value": "2017-07-29T12:00:04"
                }
            ],
            "http://example.org/common/providedBy": [
                {
                    "http://uri.etsi.org/ngsi-ld/hasObject": [
                        {
                            "@id": "urn:ngsi-ld:Person:Bob"
                        }
                    ],
                    "@type": [
                        "http://uri.etsi.org/ngsi-ld/Relationship"
                    ]
                }
            ],
            "@type": [
                "http://uri.etsi.org/ngsi-ld/Relationship"
            ]
        }
    ],
    "http://uri.etsi.org/ngsi-ld/location": [
        {
            "@type": [
                "http://uri.etsi.org/ngsi-ld/GeoProperty"
            ],
            "http://uri.etsi.org/ngsi-ld/hasValue": [
                {

```

(continues on next page)

(continued from previous page)

```

    "@value": "{ \"type\": \"Point\", \"coordinates\": [ -8.5, 41.2 ] }"
  }
]
},
"http://example.org/vehicle/speed": [
  {
    "@type": [
      "http://uri.etsi.org/ngsi-ld/Property"
    ],
    "http://uri.etsi.org/ngsi-ld/hasValue": [
      {
        "@value": 80
      }
    ]
  }
],
"@type": [
  "http://example.org/vehicle/Vehicle"
]
}'

```

c. To append additional attributes to an existing entity

POST /ngsi-ld/v1/entities/

key	Value
Content-Type	application/json
Accept	application/ld+json

Request

```

curl -iX POST \
'http://localhost:80/ngsi-ld/v1/entities/' \
-H 'Content-Type: application/json' \
-H 'Accept: application/ld+json' \
-H 'Link: <{{link}}>; rel="https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"; type="application/ld+json"' \
-d'
{
  "id": "urn:ngsi-ld:Vehicle:A100",
  "type": "Vehicle",
  ""brandName1"": {
    "type": "Property",
    "value": "BMW"
  }
}'

```

d. To update specific attributes of an existing entity**POST /ngsi-ld/v1/entities/**

key	Value
Content-Type	application/json
Accept	application/ld+json

Request

```
curl -iX POST \
'http://localhost:80/ngsi-ld/v1/entities/' \
-H 'Content-Type: application/json' \
-H 'Accept: application/ld+json' \
-H 'Link: <{{link}}>; rel="https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld";_
↪type="application/ld+json"' \
-d'
{
  "id": "urn:ngsi-ld:Vehicle:A100",
  "type": "Vehicle",

  "brandName": {
    "type": "Property",
    "value": "AUDI"
  }
}'
```

e. To delete an NGSI-LD context entity**DELETE /ngsi-ld/v1/entities/#eid**

Param	Description
eid	Entity Id

Example:

```
curl -iX DELETE http://localhost:80/ngsi-ld/v1/entities/urn:ngsi-ld:Vehicle:A100 -H
↪'Content-Type: application/json' -H 'Accept: application/ld+json'
```

f. To delete an attribute of an NGSI-LD context entity**DELETE /ngsi-ld/v1/entities/#eid/attrs/#attrName**

Param	Description
eid	Entity Id
attrName	Attribute Name

Example:

```
curl -iX DELETE http://localhost:80/ngsi-ld/v1/entities/urn:ngsi-ld:Vehicle:A100/attrs/
↪ brandName1
```

g. To retrieve a specific entity

GET /ngsi-ld/v1/entities/#eid

Param	Description
eid	Entity Id

Example:

```
curl http://localhost:80/ngsi-ld/v1/entities/urn:ngsi-ld:Vehicle:A4569
```

9.4.2 Subscription API

A new subscription is issued by the subscriber which is enrounted to broker where the details of subscriber is stored for notification purpose. The broker initiate a request to Fogflow Discovery, where this is registered as new subscription and looks for availablity of corresponding data. On receiving data it passes the information back to subscribing broker.

a. To create a new Subscription with context in Link header

POST /ngsi-ld/v1/subscriptions

Header Format

key	Value
Content-Type	application/ld+json
Link	<{{link}}>; rel="https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"; type="application/ld+json"

Request

```
curl -iX POST\
'http://localhost:80/ngsi-ld/v1/subscriptions/' \
-H 'Content-Type: application/ld+json' \
-H 'Link: <{{link}}>; rel="https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"; type="application/ld+json"' \
-d '{
  {
    "type": "Subscription",
    "id" : "urn:ngsi-ld:Subscription:71",
    "entities": [{
      "id": "urn:ngsi-ld:Vehicle:71",
      "type": "Vehicle"
    }],
    "watchedAttributes": ["brandName"],
    "notification": {
```

(continues on next page)

(continued from previous page)

```
        "attributes": ["brandName"],
        "format": "keyValues",
        "endpoint": {
            "uri": "http://my.endpoint.org/notify",
            "accept": "application/json"
        }
    }
}'
```

b. To retrieve all the subscriptions**GET /ngsi-ld/v1/subscriptions****Example:**

```
curl http://localhost:80/ngsi-ld/v1/subscriptions/ -H 'Accept: application/ld+json'
```

c. To retrieve a specific subscription based on subscription id**GET /ngsi-ld/v1/subscriptions/#sid**

Param	Description
sid	subscription Id

Example:

```
curl http://localhost:80/ngsi-ld/v1/subscriptions/urn:ngsi-ld:Subscription:71
```

d. To delete a specific subscription based on subscription id**DELETE /ngsi-ld/v1/subscriptions/#sid**

Param	Description
sid	subscription Id

Example:

```
curl -iX DELETE http://localhost:80/ngsi-ld/v1/subscriptions/urn:ngsi-ld:Subscription:71
```


SYSTEM SETUP

10.1 Start FogFlow Cloud node

10.1.1 Prerequisite

Here are the prerequisite commands for starting FogFlow:

1. docker
2. docker-compose

For ubuntu-16.04, you need to install docker-ce and docker-compose.

To install Docker CE, please refer to [Install Docker CE](#), required version > 18.03.1-ce;

Important: please also allow your user to execute the Docker Command without Sudo

To install Docker Compose, please refer to [Install Docker Compose](#), required version 18.03.1-ce, required version > 2.4.2

10.1.2 Fetch all required scripts

Download the docker-compose file and the configuration files as below.

```
# the docker-compose file to start all FogFlow components on the cloud node
wget https://raw.githubusercontent.com/smartfog/fogflow/master/release/3.2.8/cloud/
↪ docker-compose.yml

# the configuration file used by all FogFlow components
wget https://raw.githubusercontent.com/smartfog/fogflow/master/release/3.2.8/cloud/
↪ config.json
```

10.1.3 Change the IP configuration accordingly

You need to change the following IP addresses in config.json according to your own environment.

- **my_hostip**: the IP of the FogFlow cloud node and this IP address should be accessible to the FogFlow edge node. Please DO NOT use “127.0.0.1” for this.
- **site_id**: each FogFlow node (either cloud node or edge node) requires to have a unique string-based ID to identify itself in the system;
- **physical_location**: the geo-location of the FogFlow node;
- **worker.capacity**: it means the maximal number of docker containers that the FogFlow node can invoke;

Important: please DO NOT use “127.0.0.1” as the IP address of **my_hostip**, because they will be used by a running task inside a docker container.

Firewall rules: to make your FogFlow web portal accessible via the external_ip; the following ports must be open as well: 80 and 5672 for TCP

10.1.4 Start all components on the FogFlow Cloud Node

Pull the docker images of all FogFlow components and start the FogFlow system

```
# if you already download the docker images of FogFlow components, this command can
↳ fetch the updated images
    docker-compose pull

    docker-compose up -d
```

10.1.5 Validate your setup

There are two ways to check if the FogFlow cloud node is started correctly:

- Check all the containers are Up and Running using “docker ps -a”

```
docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED
↳ STATUS	PORTS		
↳	NAMES		
d4fd1aee2655	fogflow/worker	"/worker"	6 seconds ago
↳ Up 2 seconds			
↳	fogflow_cloud_worker_1		
428e69bf5998	fogflow/master	"/master"	6 seconds ago
↳ Up 4 seconds	0.0.0.0:1060->1060/tcp		
↳	fogflow_master_1		
9da1124a43b4	fogflow/designer	"node main.js"	7 seconds ago
↳ Up 5 seconds	0.0.0.0:1030->1030/tcp, 0.0.0.0:8080->8080/tcp		
↳	fogflow_designer_1		
bb8e25e5a75d	fogflow/broker	"/broker"	9 seconds ago
↳ Up 7 seconds	0.0.0.0:8070->8070/tcp		
↳	fogflow_cloud_broker_1		

(continues on next page)

(continued from previous page)

7f3ce330c204	rabbitmq:3	"docker-entrypoint.s..."	10 seconds ago	⌵
↪ Up 6 seconds	4369/tcp, 5671/tcp, 25672/tcp, 0.0.0.0:5672->5672/tcp			⌵
↪	fogflow_rabbitmq_1			
9e95c55a1eb7	fogflow/discovery	"/discovery"	10 seconds ago	⌵
↪ Up 8 seconds	0.0.0.0:8090->8090/tcp			⌵
↪	fogflow_discovery_1			

Important: if you see any container is missing, you can run “docker ps -a” to check if any FogFlow component is terminated with some problem. If there is, you can further check its output log by running “docker logs [container ID]”

10.1.6 Try out existing IoT services

Once the FogFlow cloud node is set up, you can try out some existing IoT services without running any FogFlow edge node. For example, you can try out a simple fog function as below.

- Click “Operator Registry” in the top navigator bar to trigger the initialization of pre-defined operators.

After you first click “Operator Registry”, a list of pre-defined operators will be registered in the FogFlow system. With a second click, you can see the refreshed list as shown in the following figure.

The screenshot shows the FogFlow web interface with the 'Operator Registry' tab selected. On the left, there are two input fields: 'Operator' and 'Docker Image'. A 'register' button is located below these fields. The main area displays a table titled 'list of all registered operators'.

Operator	Description	#Parameters
nodejs		0
python		0
iotagent		0
counter		0
anomaly		0
facefinder		0
connectedcar		0
recommender		0
privatesite		0
publicsite		0
pushbutton		0
acoustic		0
speaker		0
dummy		0
geohash		0

- Click “Service Topology” in the top navigator bar to trigger the initialization of pre-defined service topologies.

After you first click “Service Topology”, a list of pre-defined topologies will be registered in the FogFlow system. With a second click, you can see the refreshed list as shown in the following figure.

- Click “Fog Function” in the top navigator bar to trigger the initialization of pre-defined fog functions.

After you first click “Fog Function”, a list of pre-defined functions will be registered in the FogFlow system. With a second click, you can see the refreshed list as shown in the following figure.

- Create an IoT device entity to trigger the Fog Function

You can register a device entity via the device registration page: 1) click “System Status”; 2) click “Device”; 3) click “Add”;

FogFlow System Status Operator Registry **Service Topology** Fog Function NGSI-v1 Apps NGSI-LD Apps

list of all registered service topologies

Service Topology

Service Intent

Task Instance

register

Name	Description	#Tasks	Actions
anomaly-detection	detect anomaly events in shops	2	view delete
anomaly-detection.Id	detect anomaly events in shops	2	view delete
child-finder	search for a lost child based on face recognition	1	view delete
Crop_Prediction	This is a ML based approach for crop selection in farming	1	view delete
Heart_Health_Predictor	ML based health prediction of Human Heart	2	view delete
Id-child-finder		1	view delete

FogFlow System Status Operator Registry Service Topology **Fog Function** NGSI-v1 Apps NGSI-LD Apps

list of all registered fog functions

Fog Function

Task Instance

register

ID	Name	Action	Topology
FogFunction.Convert3	Convert3	view delete	{ "name": "Convert3", "description": "test", "tasks": [{ "name": "Main", "operator": "converter", "input_streams": [{ "selected_type": "ConnectedCar", "selected_attributes": [], "groupby": "ALL", "scoped": false }], "output_streams": [{ "entity_type": "RainObservation" }] }] }
FogFunction.Convert2	Convert2	view delete	{ "name": "Convert2", "description": "test", "tasks": [{ "name": "Main", "operator": "geohash", "input_streams": [{ "selected_type": "SmartAwning", "selected_attributes": [], "groupby": "ALL", "scoped": false }], "output_streams": [] }] }
FogFunction.Prediction	Prediction	view delete	{ "name": "Prediction", "description": "test", "tasks": [{ "name": "Main", "operator": "predictor", "input_streams": [{ "selected_type": "RainObservation", "selected_attributes": [], "groupby": "ALL", "scoped": false }], "output_streams": [{ "entity_type": "Prediction" }] }] }
FogFunction.Prediction	Prediction	view delete	{ "name": "Prediction", "description": "test", "tasks": [{ "name": "Main", "operator": "predictor", "input_streams": [{ "selected_type": "RainObservation", "selected_attributes": [], "groupby": "ALL", "scoped": false }], "output_streams": [{ "entity_type": "Prediction" }] }] }

Then you will see the following device registration page.

- Check if the fog function is triggered

Check if a task is created under “Task” in System Management.**

ID	Service	Task	Type	Worker	port	status
Task.826344878	Task	Dummy	Dummy	Worker.001	40929	running

Check if a Stream is created under “Stream” in System Management.**

ID	Entity Type	Attributes
Result.Temperature.temp001	Result	{ "DeviceID": { "type": "string", "value": "temp001", "uri": { "type": "string", "value": "http://designer:8080/photo/null" }, "iconURL": { "type": "string", "value": "/photo/default/icon.png" } } }

10.2 Start FogFlow edge node

Typically, an FogFlow edge node needs to deploy a Worker, an IoT broker and a system monitoring agent metricbeat. The Edge IoT Broker at the edge node can establish the data flows between all task instances launched on the same edge node. However, this Edge IoT Broker is optional, especially when the edge node is a very constrained device that can only support a few tasks without any data dependency.

Here are the steps to start an FogFlow edge node:

10.2.1 Install Docker Engine

To install Docker CE and Docker Compose, please refer to [Install Docker CE and Docker Compose on Raspberry Pi](#).

Note: Docker engine must be installed on each edge node, because all task instances in FogFlow will be launched within a docker container.

10.2.2 Download the deployment script

```
# the docker-compose file to start all FogFlow components on the edge node
wget https://raw.githubusercontent.com/smartfog/fogflow/master/release/3.2.8/edge/docker-
↪compose.yml

#download the deployment scripts
wget https://raw.githubusercontent.com/smartfog/fogflow/master/release/3.2.8/edge/start.
↪sh
wget https://raw.githubusercontent.com/smartfog/fogflow/master/release/3.2.8/edge/stop.sh

#make them executable
chmod +x start.sh stop.sh
```

10.2.3 Download the default configuration file

```
#download the configuration file
wget https://raw.githubusercontent.com/smartfog/fogflow/master/release/3.2.8/edge/config.
↪json
```

10.2.4 Change the configuration file accordingly

You can use the default setting for a simple test, but you need to change the following addresses according to your own environment:

- **coreservice_ip**: please refer to the configuration of the cloud part. This is the accessible address of your FogFlow core services running in the cloud node;
- **external_hostip**: this is the external IP address, accessible for the cloud broker. It is useful when your edge node is behind NAT;
- **my_hostip** is the IP of your default docker bridge, which is the “docker0” network interface on your host.

- **site_id** is the user-defined ID for the edge Node. Broker and Worker IDs on that node will be formed according to this Site ID.
- **container_autoremove** is used to configure that the container associated with a task will be removed once its processing is complete.
- **start_actual_task** configures the Fogflow worker to include all those activities that are required to start or terminate a task or maintain a running task along with task configurations instead of performing the minimal effort. It is recommended to keep it true.
- **capacity** is the maximum number of docker containers that the FogFlow node can invoke. The user can set the limit by considering resource availability on a node.

```
//you can see the following part in the default configuration file
{
  "coreservice_ip": "155.54.239.141",
  "external_hostip": "35.234.116.177",
  "my_hostip": "172.17.0.1",

  "site_id": "002",

  "worker": {
    "container_autoremove": false,
    "start_actual_task": true,
    "capacity": 4
  }
}
```

10.2.5 Start Edge node components

Note: the edge node is ARM-based, such as Raspberry Pi.

```
#start both components in the same script
./start.sh
```

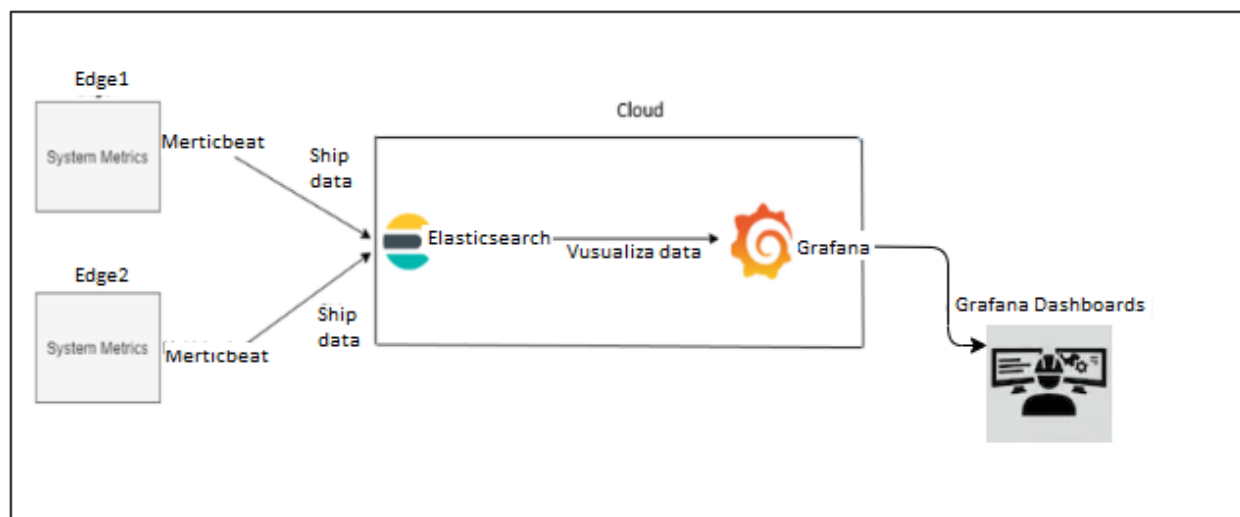
10.2.6 Stop Edge node components

```
#stop both components in the same script
./stop.sh
```


MONITORING

Fogflow system health can be monitored by system monitoring tools Metricbeat, Elasticsearch and Grafana in short EMG. With these tools edges and Fogflow Docker service health can be monitored. Metricbeat deployed on Edge node. Elasticsearch and Grafana on Cloud node.

As illustrated in following picture, set up System Monitoring tools to monitor system resource usage.



11.1 Set up Monitoring components on Cloud node

11.1.1 Fetch all required scripts

Download the docker-compose file and the configuration files as below.

```
# the docker-compose file to start all Monitoring components on the cloud node
wget https://raw.githubusercontent.com/smartfog/fogflow/master/docker/core/http/grafana/
↪ docker-compose.yml

# the configuration file used by Grafana
wget https://raw.githubusercontent.com/smartfog/fogflow/master/docker/core/http/grafana/
↪ grafana.yaml

# the configuration file used by metric beat
```

(continues on next page)

(continued from previous page)

```
wget https://raw.githubusercontent.com/smartfog/fogflow/master/docker/core/http/grafana/
↪metricbeat.docker.yml
```

JSON files to configure grafana dashboard

```
wget https://raw.githubusercontent.com/smartfog/fogflow/master/docker/core/http/grafana/
↪dashboards.zip
```

install unzip tool on system to extract JSON files from dashboards.zip

```
#command to install unzip in ubuntu
```

```
apt-get install unzip
```

```
#command to unzip the file dashboards.zip
```

```
unzip dashboards.zip
```

Note: It is supposed that FogFlow cloud components are in running state before setting up system monitoring.

11.1.2 Start all Monitoring components

```
docker-compose up -d
```

```
#Check all the containers are Up and Running using "docker ps -a"
```

```
docker ps -a
```

11.2 Configure Elasticsearch on Grafana Dashboard

Grafana dashboard can be accessible on web browser via the URL: http://<Cloud_Public_IP>:3003/. The default username and password for Grafana login are admin and admin respectively.

- After successful login to grafana, click on “Add data source” in Home Dashboard to setup the source of data.
 - Select Elasticsearch from Add Data Source page. Now the new page is Data Sources/Elasticsearch same as below figure.
1. Put a name for the Data Source i.e. “Elasticsearch”.
 2. In HTTP detail ,mention URL of your elasticsearch and Port. URL shall include HTTP for eg: “<http://192.168.100.112:9200>”
 3. In Access select “Server(default)”. URL needs to be accessible from the Grafana backend/server.
 4. In Elasticsearch details, put “@timestamp” for Time field name.
 5. Select Elasticsearch Version i.e. “7.0+”.

Then click on “Save & Test” button.

On successful configuration the dashboard will return “Index OK. Time field name OK.”

Data Sources / Elasticsearch
Type: Elasticsearch

Settings

Name Default ☒

HTTP

URL

Access [Help](#)

Whitelisted Cookies [Add](#)

Auth

Basic auth ☐ With Credentials ☐

TLS Client Auth ☐ With CA Cert ☐

Skip TLS Verify ☐

Forward OAuth Identity ☐

Elasticsearch details

Index name Pattern

Time field name

Version

Max concurrent Shard Requests

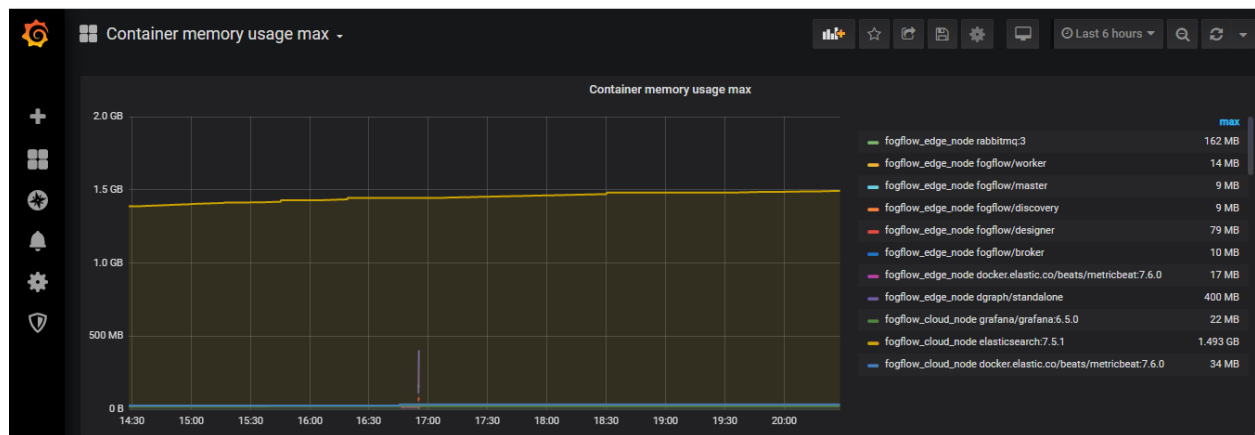
11.3 Grafana-based monitoring

Grafana based system metrics can be seen on grafana dashboard. Follow the steps:

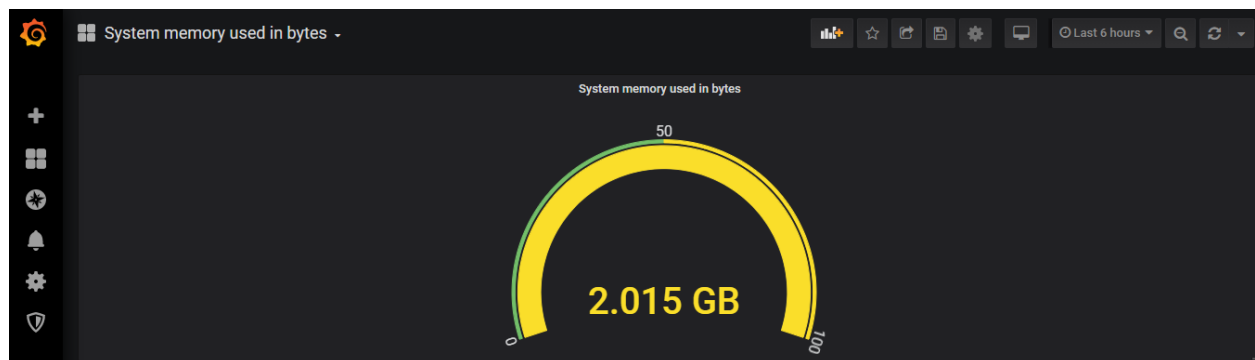
1. In the sidebar, take the cursor over Dashboards (squares) icon.
2. click Manage.
3. The dashboard appears in a Services folder.

Select particular dashboard to see the corresponding monitoring metrics.

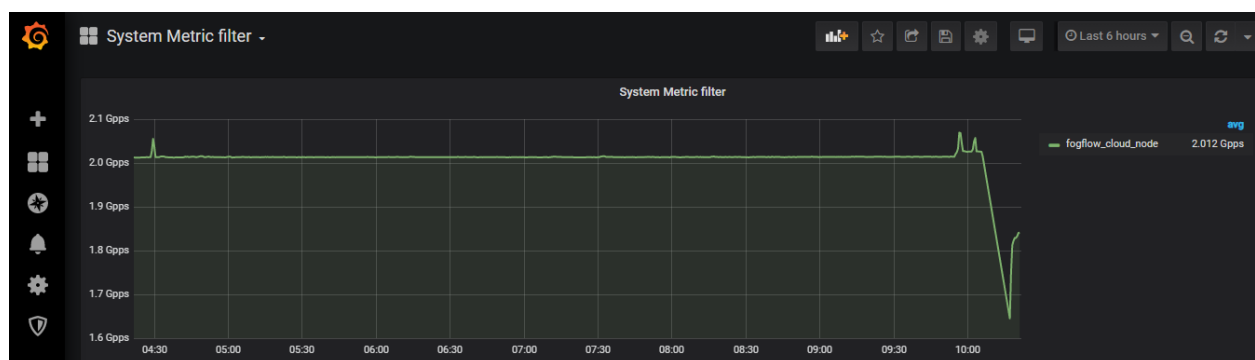
- Below dashboard diagram for containers list with maximum memory usage.



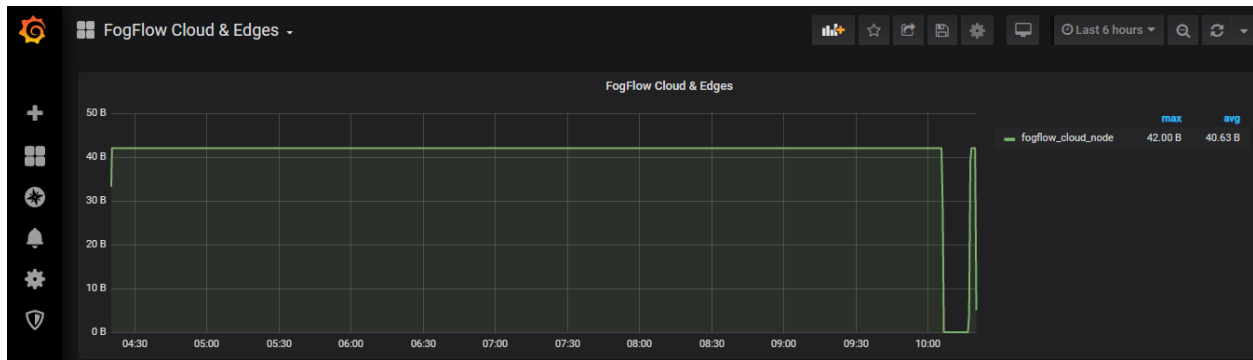
- Below dashboard diagram to show system memory used in bytes on cloud node.



- Below dashboard diagram to show system metric data rate in packet per second on cloud node.



- Below dashboard diagram to show FogFlow Cloud node that are live.



Note: Before proceeding please clear the browser cache, browser might saves some information from websites in its cache and cookies. Clearing them fixes certain problems, like loading or formatting issues on sites.

11.3.1 Set up Metricbeat on Edge node

Download the metricbeat.yml file for edge node.

```
# the configuration file used by metric beat
wget https://raw.githubusercontent.com/smartfog/fogflow/master/docker/core/http/grafana/
↪metricbeat.docker.yml
```

Optional - Edit "name" in metricbeat.docker.yml file to add particular name for better identification of edge node. Further user can update the output.elasticsearch.hosts in the metricbeat.docker.yml file.

```
name: "<155.54.239.141/edge02>"
metricbeat.modules:
- module: docker
  #Docker module parameters to monitor based on user requirement,example as below
  metricsets: ["cpu","memory","network"]
  hosts: ["unix:///var/run/docker.sock"]
  period: 10s
  enabled: true
- module: system
  #System module parameters to monitor based on user requirement, example as below
  metricsets: ["cpu","load","memory","network"]
  period: 10s

# User can update this while executing docker run command also.
output.elasticsearch:
  hosts: '155.54.239.141:9200'
```

Copy below Docker run command, edit and replace <Cloud_Public_IP> with IP/URL of elasticsearch in output.elasticsearch.hosts=["<Cloud_Public_IP>:9200"]. This command will deploy metric beat on edge node.

```
docker run -d --name=metricbeat --user=root --volume="$(pwd)/metricbeat.docker.
↪yml:/usr/share/metricbeat/metricbeat.yml:ro" --volume="/var/run/docker.sock:/var/run/
↪docker.sock:ro" --volume="/sys/fs/cgroup:/hostfs/sys/fs/cgroup:ro" --volume="/
```

(continues on next page)

(continued from previous page)

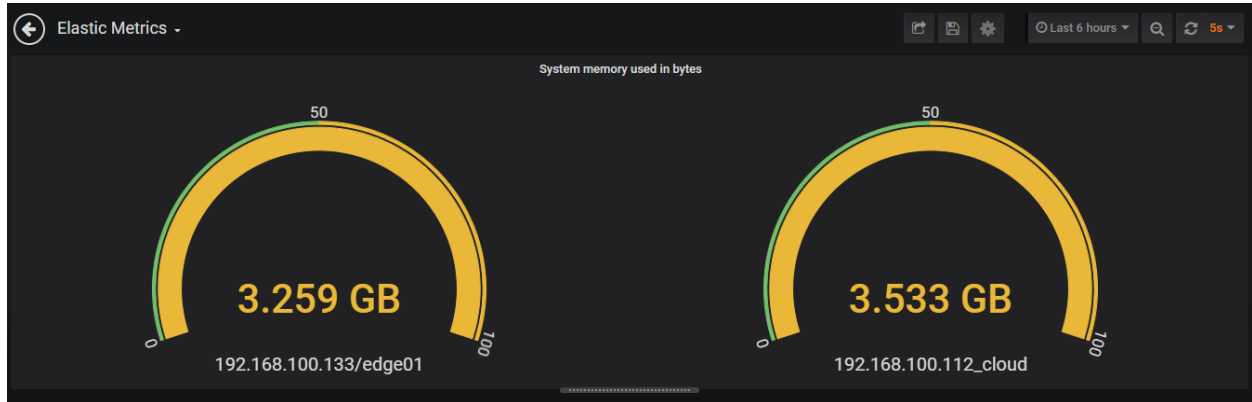
```

→proc:/hostfs/proc:ro" --volume="/:/hostfs:ro" docker.elastic.co/beats/metricbeat:7.
→6.0 metricbeat -e -E output.elasticsearch.hosts=["<Cloud_Public_IP>:9200"]

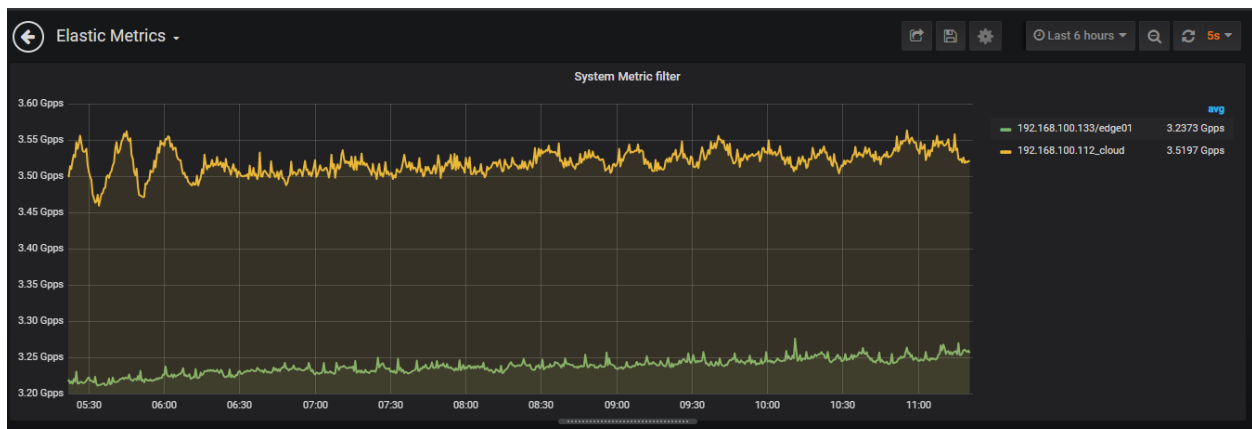
```

Metrics for Edge node can be seen on same Grafana dashboard with cloud node metrics via URL: http://<Cloud_Public_IP>:3003/.

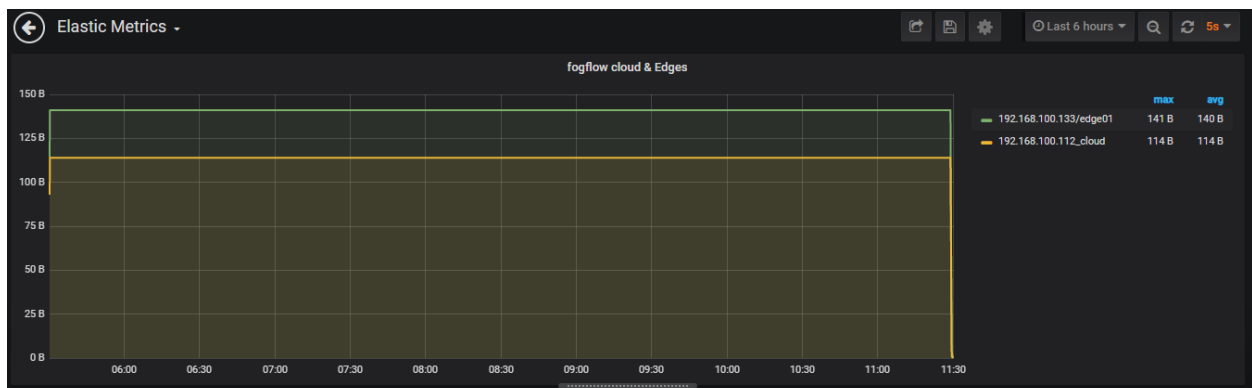
- Below dashboard diagram to show system memory used in bytes on cloud as well as on edge node.



- Below dashboard diagram to show system metric data rate in packet per second on cloud as well as on edge node.



- Below dashboard diagram to show FogFlow Cloud and Edge nodes that are live.



12.1 HTTPs-based communication

12.1.1 Secure the cloud-edge communication

To secure the communication between the FogFlow cloud node and the FogFlow edge nodes, FogFlow can be configured to use HTTPs for the NGSI9 and NGSI10 communication, which is mainly for data exchange between cloud node and edge nodes, or between two edge nodes. Also, the control channel between Topology Master and Worker can be secured by enabling TLS in RabbitMQ. The introduction steps to secure the data exchange between one FogFlow cloud node and one FogFlow edge node.

12.1.2 Configure DNS server

As illustrated by the following picture, in order to set up FogFlow to support the HTTPs-based communication, the FogFlow cloud node and the FogFlow edge node are required to have their own domain names, because their signed certificates must be associated with their domain namers. Therefore, DNS service is needed to be used to resolve the domain names for both the cloud node and the edge node. For example, [freeDNS](#) can be used for this purpose.

Important: please make sure that the domain names of the cloud node and the edge node can be properly resolved and correct IP address can be seen.

12.1.3 Set up the FogFlow cloud node

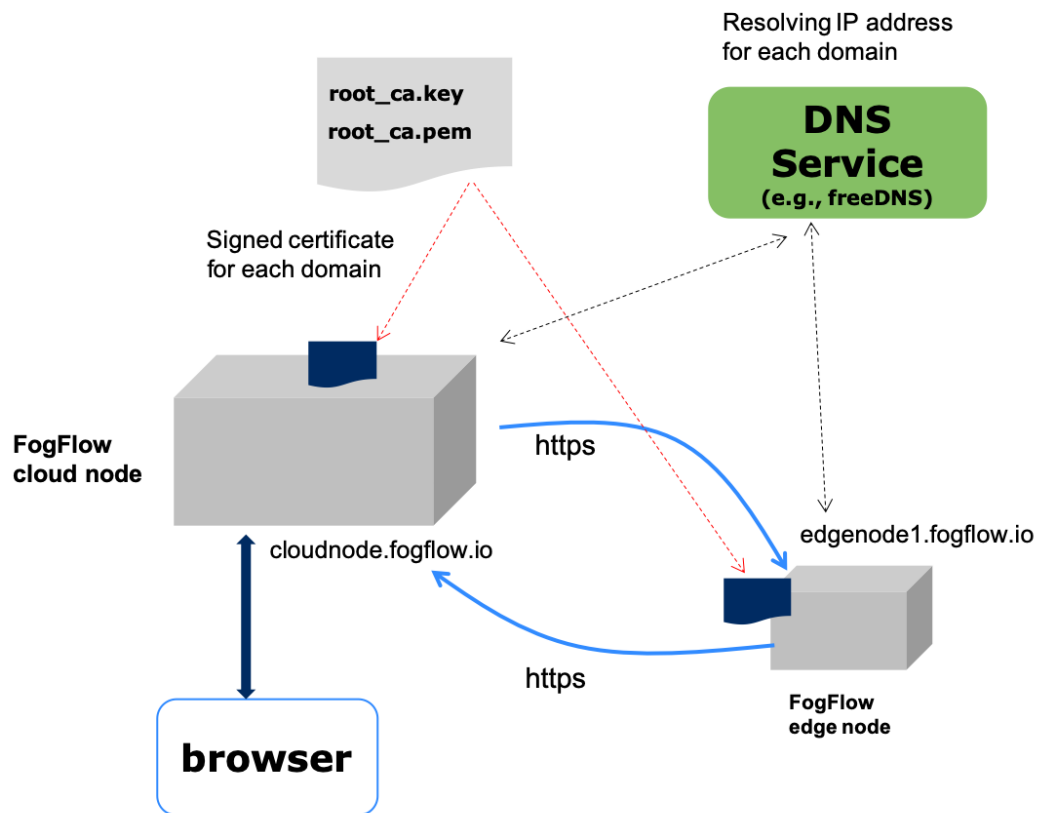
12.1.4 Fetch all required scripts

Download the docker-compose file and the configuration files as below.

```
# download the script that can fetch all required files
wget https://raw.githubusercontent.com/smartsfog/fogflow/master/docker/core/https/fetch.sh

# make this script executable
chmod +x fetch.sh

# run this script to fetch all required files
./fetch.sh
```



12.1.5 Change the configuration file

```
{
  "coreservice_ip": "cloudnode.fogflow.io",    #change this to the domain name of your
↳own cloud node
  "external_hostip": "cloudnode.fogflow.io",  #change this to the domain name of your
↳own cloud node
  ...
}
```

12.1.6 Generate the key and certificate files

```
# make this script executable
chmod +x key4cloudnode.sh

# run this script to fetch all required files
./key4cloudnode.sh cloudnode.fogflow.io
```

12.1.7 Start the FogFlow components on the cloud node

```
docker-compose up -d
```

12.1.8 Validate setup

```
docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	NAMES
90868b310608	nginx:latest	"nginx -g 'daemon of...'"	5 seconds ago	
↳ Up 3 seconds	0.0.0.0:80->80/tcp			
↳ fogflow_nginx_1				
d4fd1aee2655	fogflow/worker	"/worker"	6 seconds ago	
↳ Up 2 seconds				fogflow_
↳ cloud_worker_1				
428e69bf5998	fogflow/master	"/master"	6 seconds ago	
↳ Up 4 seconds	0.0.0.0:1060->1060/tcp			fogflow_
↳ master_1				
9da1124a43b4	fogflow/designer	"node main.js"	7 seconds ago	
↳ Up 5 seconds	0.0.0.0:1030->1030/tcp, 0.0.0.0:8080->8080/tcp			fogflow_
↳ designer_1				
bb8e25e5a75d	fogflow/broker	"/broker"	9 seconds ago	
↳ Up 7 seconds	0.0.0.0:8070->8070/tcp			fogflow_
↳ cloud_broker_1				
7f3ce330c204	rabbitmq:3	"docker-entrypoint.s..."	10 seconds ago	
↳ Up 6 seconds	4369/tcp, 5671/tcp, 25672/tcp, 0.0.0.0:5672->5672/tcp			
↳ fogflow_rabbitmq_1				
9e95c55a1eb7	fogflow/discovery	"/discovery"	10 seconds ago	
↳ Up 8 seconds	0.0.0.0:8090->8090/tcp			fogflow_

(continues on next page)

(continued from previous page)

```
↪discovery_1
   399958d8d88a      grafana/grafana:6.5.0   "/run.sh"           29 seconds ago
↪  Up 27 seconds    0.0.0.0:3003->3000/tcp           fogflow_
↪grafana_1
   9f99315a1a1d      fogflow/elasticsearch:7.5.1 "/usr/local/bin/dock..." 32 seconds
↪ago  Up 29 seconds  0.0.0.0:9200->9200/tcp, 0.0.0.0:9300->9300/tcp
↪fogflow_elasticsearch_1
   57eac616a67e      fogflow/metricbeat:7.6.0   "/usr/local/bin/dock..." 32 seconds ago
↪  Up 29 seconds
↪fogflow_metricbeat_1
```

12.1.9 Set up the FogFlow edge node

12.1.10 Fetch all required scripts

Download the docker-compose file and the configuration files as below.

```
# download the script that can fetch all required files
wget https://raw.githubusercontent.com/smartfog/fogflow/master/docker/edge/https/fetch.sh

# make this script executable
chmod +x fetch.sh

# run this script to fetch all required files
./fetch.sh
```

12.1.11 Change the configuration file

```
{
  "coreservice_ip": "cloudnode.fogflow.io",  #change this to the domain name of your
↪own cloud node
  "external_hostip": "edgenode1.fogflow.io", #change this to the domain name of your
↪own edge node
  ...
}
```

12.1.12 Generate the key and certificate files

```
# make this script executable
chmod +x key4edgenode.sh

# run this script to fetch all required files
./key4edgenode.sh  edgenode1.fogflow.io
```

12.1.13 Start the FogFlow components on the edge node

```
docker-compose up -d
```

12.1.14 Validate setup

```
docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
16af186fb54e	fogflow/worker	"/worker"	About a minute ago	Up
195bb8e44f5b	fogflow/broker	"/broker"	About a minute ago	Up

12.1.15 Check system status via FogFlow Dashboard

FogFlow dashboard can be opened in web browser to see the current system status via the URL: <https://cloudnode.fogflow.io/index.html>

Important: please make sure that the domain names of the cloud node can be properly resolved.

If self-signed SSL certificate is being used, a browser warning indication can be seen that the certificate should not be trusted. It can be proceeded past this warning to view the FogFlow dashboard web page via https.

12.2 Secure FogFlow using Identity Management

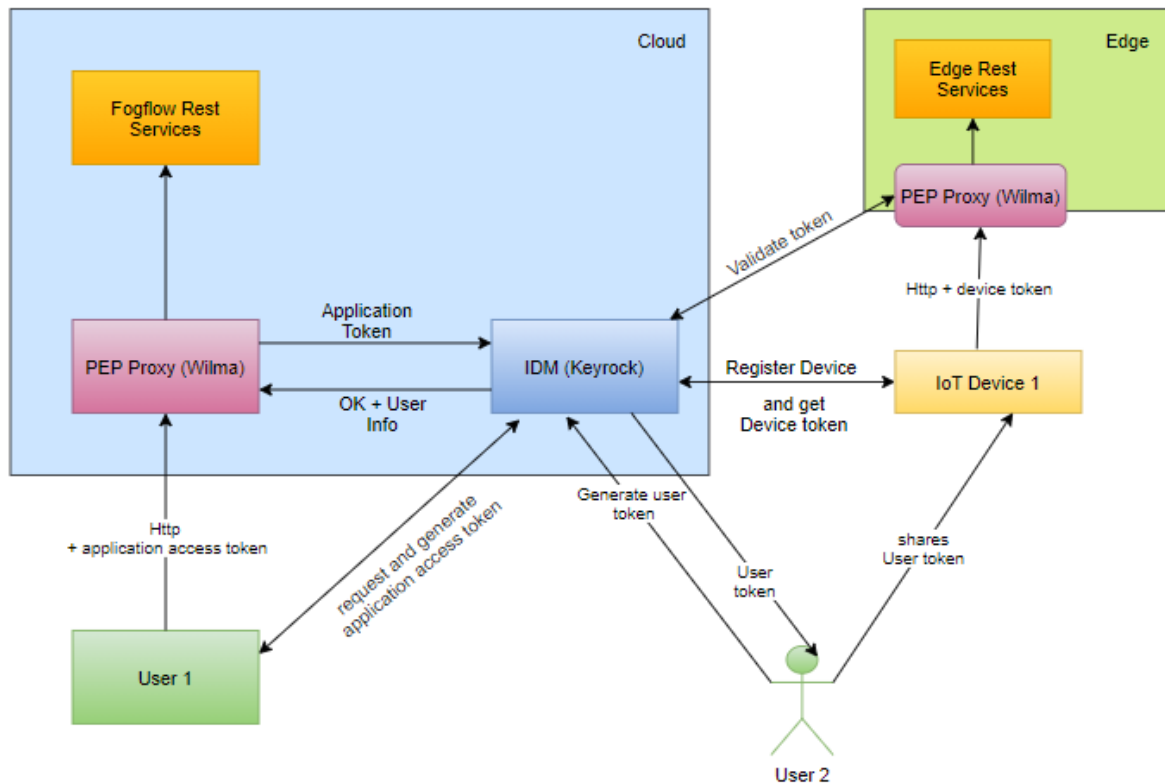
Identity management(IdM) is a process for identifying, authenticating individuals or groups to have access to applications or system by associating some auth token with established identities. IdM is the task of controlling data about users or applications. In this tutorial FogFlow Designer security implementation and secure Cloud-Edge communication is explained and tested.

12.2.1 Terminology

Keyrock: Keyrock is the FIWARE component responsible for Identity Management. Keyrock also provide feature to add OAuth2-based authentication and authorization security in order to secure services and applications.

PEP Proxy Wilma: PEP Proxy Wilma is a FIWARE Generic Enabler that enhances the performance of Identity Management. It combines with Keyrock to secure access to endpoints exposed by FIWARE Generic Enablers. Wilma listens for any request, authenticates it from Keyrock and stores it in its cache for a limited period of time. If a new request arrives, Wilma will first check in its cache and if any grant is stored, it will directly authenticate otherwise it will send the request to Keyrock for authentication.

12.2.2 Security Architecture



12.2.3 Cloud and Edge Interaction with IDM

FogFlow cloud node flow:

1. As in architecture diagram, PEP Proxy will register itself on behalf FogFlow Designer first on Keyrock. Detail explanation is given in [below](#) topics of this tutorial.
2. User can access Designer via PEP proxy proxy by using the access-token of PEP proxy in request header.

FogFlow edge node flow:

1. On behalf of edge node, one instance of PEP Proxy will be pre-registered on keyrock, edge will be using oauth credentials to fetch PEP Proxy details. Detail explanation is given in [below](#) topics of this tutorial. Click [here](#) to refer.
2. After the authentication edge node will be able to communicate with FogFlow cloud node.
3. Any device can register itself or communicate with FogFlow edge node using access-token generated on behalf of each IoT Device registered at Keyrock.

12.2.4 Installation of Security Components on Cloud

```
# the docker-compose file to start Identity Manager on the cloud node
wget https://raw.githubusercontent.com/smartfog/fogflow/development/docker/core/http/
↳security_setup/docker-compose.idm.yml

# the configuration file used by IdM
wget https://raw.githubusercontent.com/smartfog/fogflow/development/docker/core/http/
↳security_setup/idm_config.js

# the docker-compose file to start PEP Proxy (Wilma) on the cloud node
wget https://raw.githubusercontent.com/smartfog/fogflow/development/docker/core/http/
↳security_setup/docker-compose.pep.yml

# the configuration file used by PEP Proxy
wget https://raw.githubusercontent.com/smartfog/fogflow/development/docker/core/http/
↳security_setup/pep_config.js
```

12.2.5 Change the IP configuration accordingly

Configuration file need to be modified at the following places with IP addresses according to user own environment.

- Change the IdM config file (idm_config.js) at following places as per the environment.

```
config.port = 3000;
config.host = '<IdM_IP>'; // eg; config.host = '180.179.214.215';
```

Note: IdM_IP denotes the IP of cloud node in this case, if IdM instance is to be set on platform other than cloud; user must provide IP of that platform.

- If user wants to setup database according to their need, they can do so by changing following places in idm_config.js as per environment. For default usage, do not change below mentioned configuration in idm_config.js.

```
// Database info
config.database = {
host: 'localhost',
password: 'idm',
username: 'root',
database: 'idm',
dialect: 'mysql',
port: undefined
};
```

12.2.6 Start Security Components on Cloud Node

Start Identity Manager

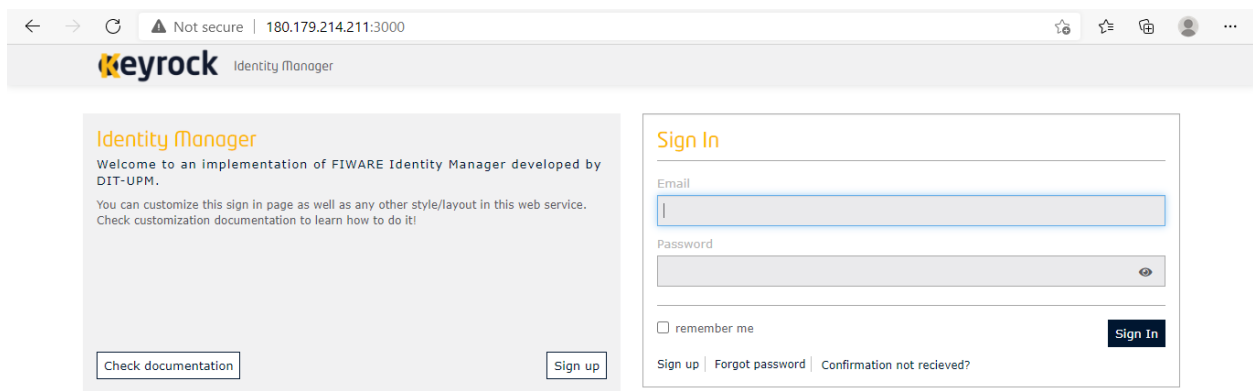
```
docker-compose -f docker-compose.idm.yml up -d
```

#Check all the containers are Up and Running using "docker ps -a"
 docker ps -a

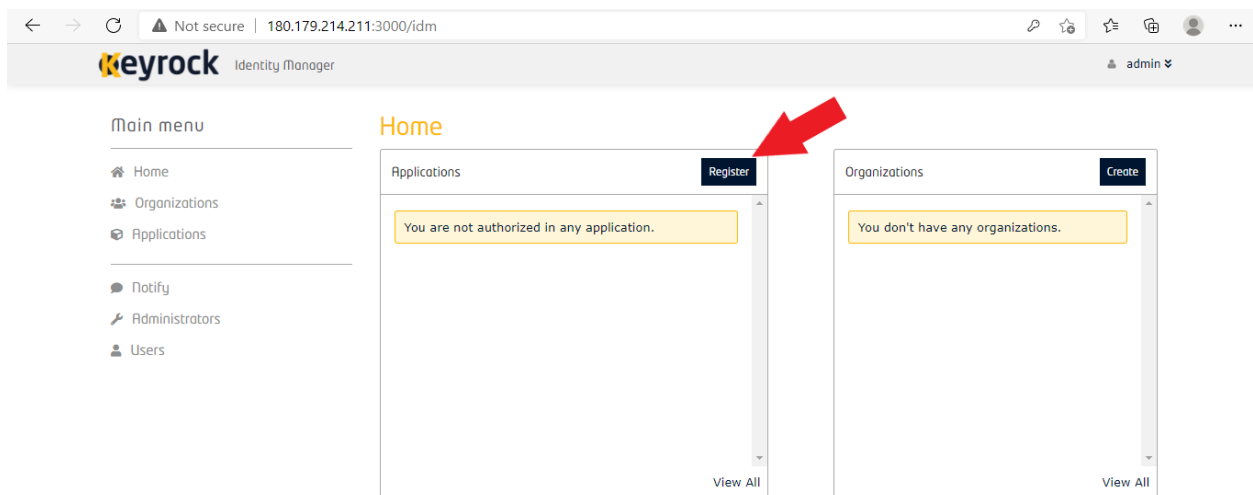
Note: IdM dashboard can be accessed on the http://<Idm_Ip>:3000 (for eg. <http://180.179.214.215:3000>) on browser.

Register Application with IdM

For accessig above dashboard, user needs to login with his credentials i.e. username and password. By default user can use admin credentials which are "admin@test.com" and "1234". After login, the below screen would appear.



- Now to register application, click on register tab under application heading.



- Now enter details as below

Name : (Provided by user)
 Description : (Provided by User)

(continues on next page)

(continued from previous page)

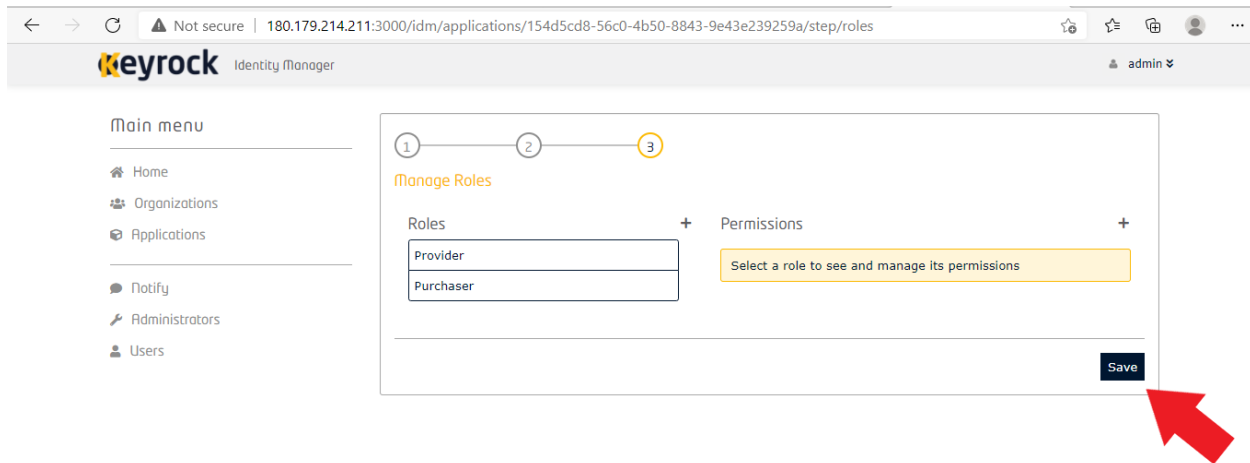
Url : (Cloud Node's Designer IP for eg: `http://180.179.214.215` where "180.179.214.215" is the IP for working cloud node)
 Callback Url : (in case of designer as an application it would be `http://180.179.214.215/index.html`)

click on Next button.

- If user wants to add image icon for his application he can do that by uploading it. Click Next button after that.

click to upload icon

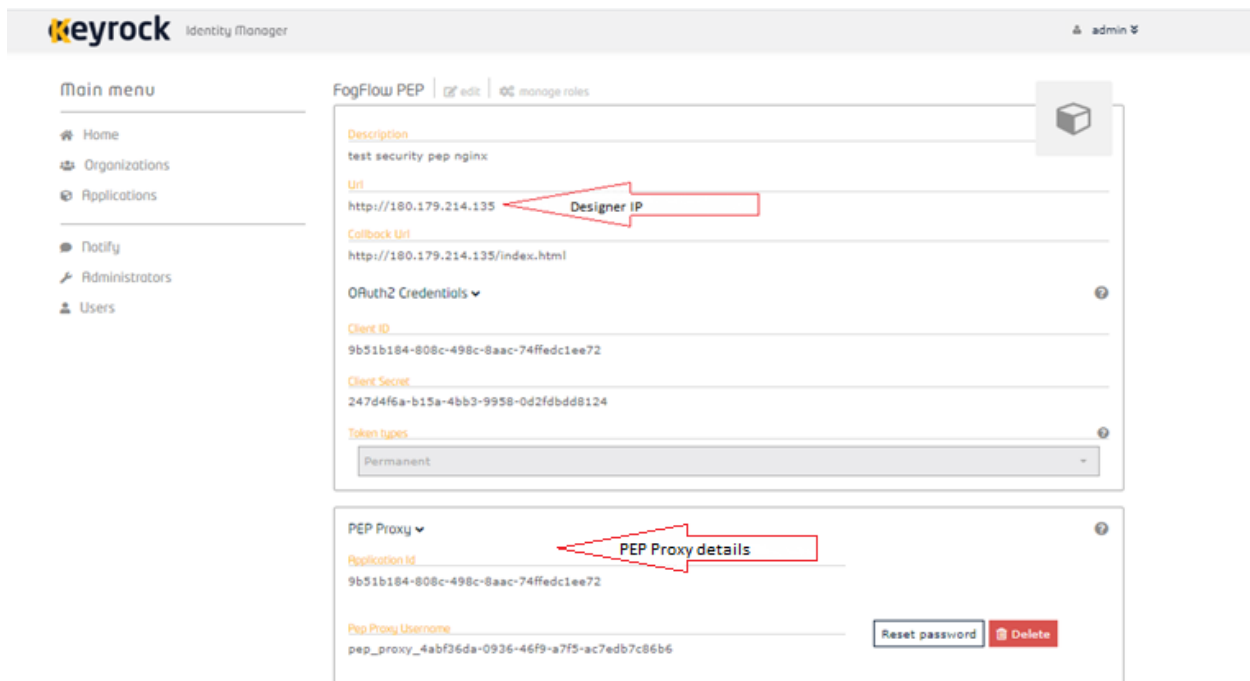
- Again click Save button to finish the registration.



12.2.7 Start PEP Proxy (Wilma) on Cloud node

Below are the steps that need to be done to setup communication between IdM and PEP Proxy.

- Authenticate PEP Proxy itself with Keyrock Identity Management.



Login to Keyrock (<http://180.179.214.135:3000/idm/>) account with user credentials i.e. Email and Password.

For Example: admin@test.com and 1234.

After Login, Click “Applications” then select the registered Application. Click “PEP Proxy” link to get Application ID , PEP Proxy Username and PEP Proxy Password.

Note: Application ID , PEP Proxy Username and PEP Proxy Password will generate by clicking ‘Register PEP Proxy’ button.

To setup PEP proxy for securing Designer, change the followings inside the pep_config file. Get PEP Proxy Credentials from Keyrock Dashboard while registering an application.

```
config.pep_port = process.env.PEP_PROXY_PORT || 80;
config.idm = {
  host: process.env.PEP_PROXY_IDM_HOST || '180.179.214.135',
  port: process.env.PEP_PROXY_IDM_PORT || 3000,
  ssl: toBoolean(process.env.PEP_PROXY_IDM_SSL_ENABLED, false),
};
config.app = {
  host: process.env.PEP_PROXY_APP_HOST || '180.179.214.135',
  port: process.env.PEP_PROXY_APP_PORT || '80',
  ssl: toBoolean(process.env.PEP_PROXY_APP_SSL_ENABLED, false), // Use true if the app
  server listens in https
};

config.pep = {
  app_id: process.env.PEP_PROXY_APP_ID || '9b51b184-808c-498c-8aac-74ffedc1ee72',
  username: process.env.PEP_PROXY_USERNAME || 'pep_proxy_4abf36da-0936-46f9-a7f5-
ac7edb7c86b6',
  password: process.env.PEP_PASSWORD || 'pep_proxy_fb4955df-79fb-4dd7-8968-e8e60e4d6159',
  token: {
    secret: process.env.PEP_TOKEN_SECRET || '', // Secret must be configured in order
    validate a jwt
  },
  trusted_apps: [],
};
```

Note: PEP_PORT should be changed by user as per need. PEP_PROXY_IDM_HOST and PEP_PROXY_IDM_PORT should match with above setup for IdM, that means PEP_PROXY_IDM_HOST should be the IP where IdM is working and PEP_PROXY_IDM_PORT be the one, on which IdM is listening. PEP_PROXY_APP_HOST is the IP of cloud node where designer is running and PEP_PROXY_APP_PORT be the one where designer is listening. PEP_PROXY_APP_ID, PEP_PROXY_USERNAME and PEP_PASSWORD is retrieved from the registered application as shown in above image.

- Now start the PEP Proxy container, as shown below

```
docker-compose -f docker-compose.pep.yml up -d

// To check the status of container, use
docker ps -a
```

12.2.8 Generate Application Access Token

Request Keyrock IDM to generate application access-token and refresh token.

1. Set the HTTP request Header, payload and Authorization field as per below screen shots.
2. Click “Send” Button to get application access-token.

Note: Above request is sent using POSTMAN Application. User can obtain Client ID and Client Secret from Keyrock dashboard. To retrieve Client ID and Client Secret, click on registered application and under ‘OAuth2 Credentials’, user can find Client ID and Client Secret.

Note: The Authorization code can be generated using below command.

```
echo -n Client_ID:Client_SECRET | base64 | tr -d " \t\n\r"
```

http://180.179.214.135:3000/oauth2/token/

POST http://180.179.214.135:3000/oauth2/token/ Send

Params Authorization Headers (1) Body Pre-request Script Tests Cookies Code

TYPE: Basic Auth

The authorization header will be automatically generated when you send the request. [Learn more about authorization](#)

Preview Request

Username: 9b51b184-808c-498c-8aac-74ffedc1ee72 (Client ID)

Password: (Client Secret)

☐ Show Password

keyrock Identity Manager admin

Main menu: Home, Organizations, Applications, Notify, Administrators, Users

FogFlow PEP edit manage roles

Description: test security pep nginx

Uri: http://180.179.214.135 (Designer IP)

Callback Uri: http://180.179.214.135/index.html

OAuth2 Credentials

Client ID: 9b51b184-808c-498c-8aac-74ffedc1ee72 (Client Id)

Client Secret: 247d4f6a-b15a-4bb3-9958-0d2fdbdd8124 (Client Secret)

Token types: Permanent

PEP Proxy

Application Id: 9b51b184-808c-498c-8aac-74ffedc1ee72 (PEP Proxy details)

PEP Proxy Username: pep_proxy_4abf36da-0936-46f9-a7f5-ac7edb7c86b6

Reset password Delete

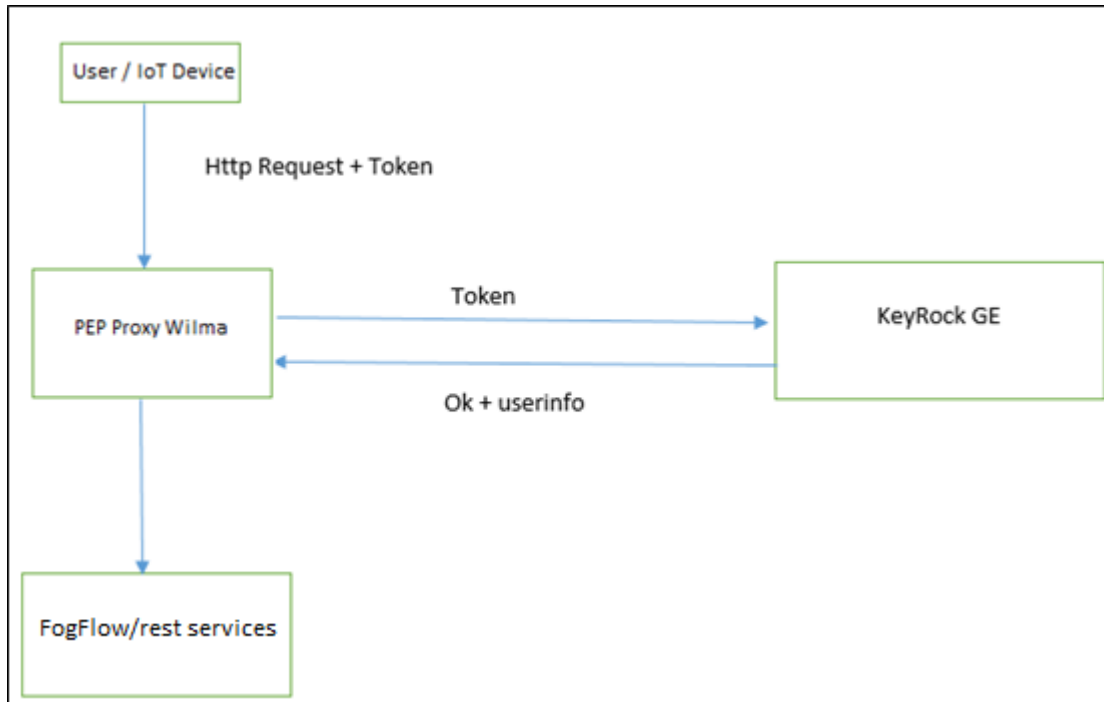
Fig. 1: Above request can be made using curl, as shown below

```
curl --request POST '<IdM_IP>:3000/oauth2/token/' \
--header 'Authorization: Basic YzNlZGU1NTU0OTIyOC00YjhlLTllNTktZTAxZWQ0Y2VhNDFjOmU4OWRlNzB1LTU3M2Q0tNDBhYS1hNjJlLWVhZDYwNGFkYTAYYw==' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'grant_type=password' \
--data-urlencode 'username=admin@test.com' \
--data-urlencode 'password=1234'
```

- The request can be made in either of the above two mentioned ways. The result will provide access token.

```
11/02/2021 15:14:33 /home/mobaxterm curl --request POST '172.30.48.46:3000/oauth2/token/' \
> --header 'Authorization: Basic YzNlZGU1NTU0OTIyOC00YjhlLTllNTktZTAxZWQ0Y2VhNDFjOmU4OWRlNzB1LTU3M2Q0tNDBhYS1hNjJlLWVhZDYwNGFkYTAYYw==' \
> --header 'Content-Type: application/x-www-form-urlencoded' \
> --header 'Cookie: session=eyJyZWVhZDYwNGFkYTAYYw==; session.sig=F-o8DxXq0zqkU1wmb09APs0KzPg' \
> --data-urlencode 'grant_type=password' \
> --data-urlencode 'username=admin@test.com' \
> --data-urlencode 'password=1234'
{"access_token":"b4ebcf801fb53756d0e0d458965e1bcd376a9331","token_type":"bearer","expires_in":3599,"refresh_token":"d2b8e9ccc8a29b97b98df88bc5e4825fcbdd65ab","scope":["bearer"]}
```

The flow of cloud security implementation can be understood by below figure.



Below are some points related to above architecture diagram:

1. Registered a PEP Proxy for designer as an application in Keyrock.
2. Keyrock will send access-token to pep.
3. Using that token user will send create entity request to designer.
4. Designer will send token to keyrock to authenticate.
5. Entity creation request will transfer to FogFlow.

entity Registration using token_access

```
curl -iX POST 'http://<Cloud_Public_IP>:<PEP_Host-port>/ngsi10/updateContext' -H
→ 'X-Auth-Token: <token>' -H 'Content-Type: application/json'
-d '
{
  "contextElements": [
    {
      "entityId": {
        "id": "Temperature100",
        "type": "Temperature",
        "isPattern": false
      },
      "attributes": [
        {
          "name": "temp",
          "type": "float",
          "value": 34
        }
      ],
      "domainMetadata": [
        {
          "name": "location",
          "type": "point",
          "value": {
            "latitude": 49.406393,
            "longitude": 8.684208
          }
        }
      ],
      "updateAction": "UPDATE"
    }
  ]
}'
```

12.2.9 Setup components on Edge

FogFlow edge node mainly contains edge broker and edge worker. To secure FogFlow edge communication between IoT device and edge node, PEP Proxy has been used. In order to create an Auth Token, firstly register an IoT device on Keyrock. So, a script will call with the start of edge node and it will instantiate a PEP Proxy with the keyrock and also setup configuration file for PEP Proxy to work, using the Keyrock APIs. The script will perform following steps:

Prerequisite

Two commands need to install before setup edge:

1. Curl
2. jq

12.2.10 scripts Installation

Below scripts need to download for setting up edge node.

```
#download the deployment scripts
wget https://raw.githubusercontent.com/smartfog/fogflow/development/docker/edge/http/
↪start.sh
wget https://raw.githubusercontent.com/smartfog/fogflow/development/docker/edge/http/
↪stop.sh
wget https://raw.githubusercontent.com/smartfog/fogflow/development/docker/edge/http/
↪script.sh
wget https://raw.githubusercontent.com/smartfog/fogflow/development/docker/edge/http/
↪oauth_config.js
wget https://raw.githubusercontent.com/smartfog/fogflow/development/docker/edge/http/pep-
↪config.js

#make them executable
chmod +x script.sh start.sh stop.sh
```

12.2.11 Change the IP configuration accordingly

Change the following things in configuration file:

- Change the oauth_config.js and add IdM IP, Edge IP which is needed to fetch configuration settings for PEP Proxy.

Start Edge node components

```
#start components in the same script
./start.sh
```

To secure FogFlow edge-IoT device communication Auth Token has been used on behalf of each IoT device. In order to create an Auth Token,

- An IoT device is needed to be registered on Keyrock.
- A script will be called with the start of edge node and it will configure PEP Proxy with keyrock on behalf of that edge node using the Keyrock APIs.

Note: the start.sh script will return Application ID, Application Secret, PEP Proxy ID, PEP Proxy Secret, Authorization code, IDM Token and the access token on console. Please save these for further use.

IoT Device Interaction with FogFlow

Flow of Requests as shown in diagram:

Step 1 : User will make a request to IDM using his credentials to generate user access token specific for that user. For this, user can use the script along with his username and password.

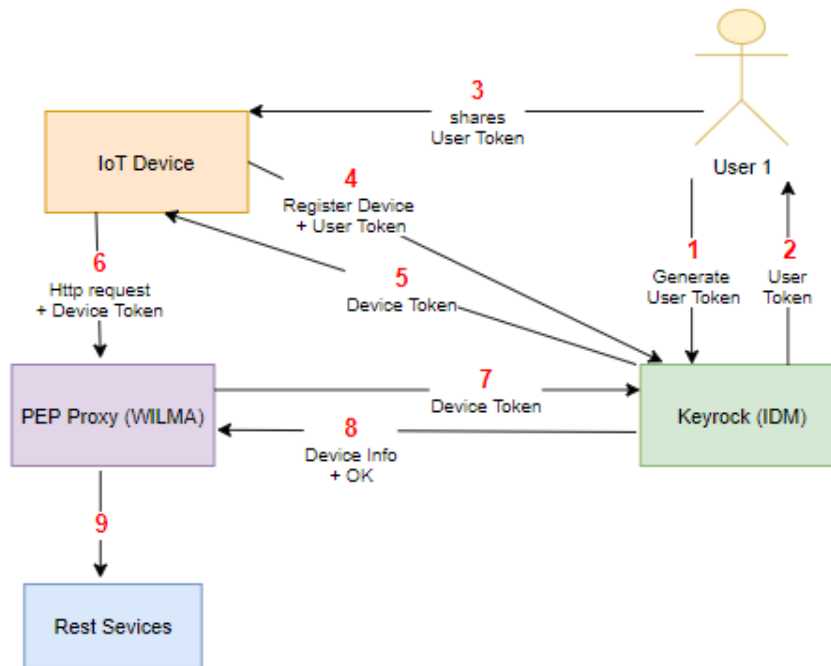
```
./user_token_generation.sh admin@test.com 1234
```

Note: For example, in above snippet admin username is “admin@test.com” and password is “1234”

Step 2 : Script will return an user access token as shown below.

Step 3 : User shares his access token (i.e. User Access Token) with IoT Device.

Step 4 : Then IoT devices get registered using the user access token passed as an argument to a script.



```

% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
   Dload  Upload  Total    Dload  Upload           Total   Spent    Left     Speed
100  199  100  138  100    61   3753   1659  --:--:--  --:--:--  --:--:--    3833
-----
IDM token is f9ffa629-9aff-4c98-ac57-1caa2917fed2
-----

```

```
./device_token_generation.sh f9ffa629-9aff-4c98-ac57-1caa2917fed2
```

Note: For example, in above snippet “f9ffa629-9aff-4c98-ac57-1caa2917fed2” is the user access token.

Step 5 : Script will return device access token and device credentials(ID and password) as shown below.

```
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
Dload  Upload  Total    Spent    Left    Speed
100  931  100  931    0    0   51351      0 --:--:-- --:--:-- --:--:-- 51722
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
Dload  Upload  Total    Spent    Left    Speed
100 1002  100 1002    0    0   49067      0 --:--:-- --:--:-- --:--:-- 50100
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
Dload  Upload  Total    Spent    Left    Speed
100  131  100  131    0    0   2904      0 --:--:-- --:--:-- --:--:-- 2977
-----
ID is iot_sensor_e8e40576-01ea-47dc-bcd6-173581c19dd4 and PASSWORD is iot_sensor_9933d670-04e9-448b-b321-4a8d28300d24
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
Dload  Upload  Total    Spent    Left    Speed
100  310  100  177  100  133   5010   3765 --:--:-- --:--:-- --:--:-- 5057
-----
Device access token is 6bf0d285d00105115063c27bf7f6135f4f080707
```

Step 6 : Now, using the above device access token, the IoT Device can interact with Edge node via making Fogflow specific requests to PEP Proxy port.

12.2.12 Register IoT Device on Keyrock Using curl request

An example request to register IoT Device is given below

```
curl --include \
  --request POST \
  --header "Content-Type: application/json" \
  --header "X-Auth-token: <token-generated-from-script>" \
  'http://keyrock/v1/applications/6e396def-3fa9-4ff9-84eb-266c13e93964/iot_agents'
```

Note: Please save the device Id and device password for further utilisation

The screenshot shows the Keyrock Identity Manager web interface. On the left is a 'Main menu' with links to Home, Organizations, Applications, Notify, Administrators, and Users. The main content area is titled 'Fogflow_PEP' and includes an 'edit' link and a 'manage roles' button. Below this, there are several configuration fields: 'Description' (TESTING), 'Url' (http://180.179.214.199), 'Callback Url' (http://180.179.214.199/index.html), and 'ORuthZ Credentials'. Below these is a 'PEP Proxy' section. At the bottom, there is a table of 'IoT Sensors' with columns for 'Id of Sensor' and 'Password of Sensor'. Two sensors are listed: one with ID 'iot_sensor_352a6227-0d06-44df-93d2-16c453b8e393' and another with ID 'iot_sensor_591d1521-a6d3-4667-a02d-0526ce9d0d95'. Each sensor entry has 'Reset password' and 'Delete' buttons. A 'Register a new IoT Sensor' button is at the bottom of the table.

An example request to generate Auth token for each registered IoT sensor is given below

```
curl -iX POST \
  'http://<IDM_IP>:3000/oauth2/token' \
  -H 'Accept: application/json' \
  -H 'Authorization: Basic <code-generated-from-script>' \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  --data "username=iot_sensor_02bc0f75-07b5-411a-8792-4381df9a1c7f&password=iot_sensor_
  ↪277bc253-5a2f-491f-abaa-c7b4e1599d6e&grant_type=password"
```

Note: Please save the Access Token for further utilisation

```
{"access_token":"5f8cbc6d8ac0d750f88220043c497feae8797d7c","token_type":"bearer","expires_in":3599,"refresh_token":"bdf09730ebc281cf1f29f
e54c6eb936fd4607c17","scope":["bearer"]}
```

12.2.13 Register Device on Edge Node using curl request

An example payload of registration device is given below.

```
Curl -iX POST 'http://<Application_IP>:<Application_Port>/NGSI9/registerContext' -H
  ↪'Content-Type: application/json' -H 'fiware-service: openiot' -H 'X-Auth-token: <token-
  ↪generated-for-IoT-device>' -H 'fiware-servicepath: /' -d '
  {
    "contextRegistrations": [
      {
        "entities": [
          {
            "type": "Lamp",
            "isPattern": "false",
            "id": "Lamp.0020"
          }
        ],
        "attributes": [
          {
            "name": "on",
            "type": "command"
          },
          {
            "name": "off",
            "type": "command"
          }
        ],
        "providingApplication": "http://0.0.0.0:8888"
      }
    ],
    "duration": "P1Y"
  }'
```

Stop Edge Node Components

- Use the below script to stop edge components that is broker and worker.

```
#stop all components in the same script
./stop.sh
```


COMPILE THE SOURCE CODE

FogFlow can be build and installed on Linux for both ARM and X86 processors (32bits and 64bits).

13.1 Install dependencies

1. To build FogFlow, first install the following dependencies.

- install git client: please follow the instruction at <https://www.digitalocean.com/community/tutorials/how-to-install-git-on-ubuntu-16-04>
- install Docker CE: please follow the instruction at <https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-16-04>

Note: all the scripts are prepared under the assumption that docker can be run without sudo.

2. To check out the code repository

```
cd /home/smartfog/go/src/  
git clone https://github.com/smartfog/fogflow.git
```

3. To build all components from the source code with multistage building

```
./build.sh multistage
```


TEST

Please follow the following steps to deploy the entire FogFlow system on a single Linux machine before your test.

[Set up all FogFlow component on a single machine](#)

Once the FogFlow is up and running, an end-to-end function test can be carried out by JMeter with a provided test plan. More detailed steps are available [here](#).

[JMeter Test](#)

RELATED PUBLICATIONS

1. F. Cirillo, B. Cheng, R. Porcellana, M. Russo, G. Solmaz, H. Sakamoto, and S. P. Romano. “[IntentKeeper: Intent-Oriented Data Usage Control for Federated Data Analytics](#), ” In IEEE LCN’20, November 2020.
2. B. Cheng, J. Fürst, G. Solmaz, T. Sanada, “[Fog Function: Serverless Fog Computing for Data Intensive IoT Services](#),” in the proceedings of 2019 IEEE Conference on Service Computing (IEEE SCC’19) (**Won the best paper award**), Milan, 2019, pp.28-35
3. M. Fadel Argerich, B. Cheng, J. Fuerst, “[Reinforcement Learning based Orchestration for Elastic Services](#)”, in the proceedings of the 5th IEEE World Forum on Internet of Things, WF-IoT 2019, Limerick, Ireland, April 15-18, 2019, pp. 352-357
4. B. Cheng, E. Kovacs, A. Kitazawa, K. Terasawa, T. Hada, M. Takeuchi, “[FogFlow: Orchestrating IoT Services over Cloud and Edges](#)”, NEC Technical Journal, 2018/11
5. B. Cheng, G. Solmaz, F. Cirillo, E. Kovacs, K. Terasawa and A. Kitazawa, “[FogFlow: Easy Programming of IoT Services Over Cloud and Edges for Smart Cities](#)”, in IEEE Internet of Things Journal, 2017 (**Won the Best Paper Runner-up award**)
6. B. Cheng, A. Papageorgiou and M. Bauer, “[Geelytics: Enabling On-Demand Edge Analytics over Scoped Data Sources](#)”, 2016 IEEE International Congress on Big Data (IEEE BigData Congress), San Francisco, CA, 2016, pp. 101-108
7. B. Cheng, A. Papageorgiou, F. Cirillo and E. Kovacs, “[GeeLytics: Geo-distributed Edge Analytics for Large Scale IoT Systems Based on Dynamic Topology](#)”, IEEE 2nd World Forum on Internet of Things (WF-IoT), Milan, 2015, pp. 565-570.

TROUBLESHOOTING

This document discusses some common problems that people run into when using FogFlow as well as some known problems. If you encounter other problems, please *let us know*. .. let us know: <https://github.com/smartfog/fogflow/issues>

- **edge node is behind NAT**

If your edge node is behind NAT or a firewall that blocks any incoming notify to your IoT Broker at edge, your edge node can not work properly. We are going to support this type of setup in the near future.

CONTACT

The following are good places to discuss FogFlow.

1. [Our Mailing List](#): Sending us an email to discuss anything related to development, usage, or other general questions.
2. [FIWARE Q&A](#): To discuss any question or issue with other FIWARE users.
3. [GitHub Issues](#): For bug reports and feature requests.