# Quick Start

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>High level introduction</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>One-page tutorial</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>Set up everything on a single machine</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>Define and trigger a fog function</td>
<td>19</td>
</tr>
<tr>
<td>5</td>
<td>Define and trigger a service topology</td>
<td>31</td>
</tr>
<tr>
<td>6</td>
<td>Connect an IoT device to FogFlow</td>
<td>45</td>
</tr>
<tr>
<td>7</td>
<td>Integrate FogFlow with FIWARE</td>
<td>49</td>
</tr>
<tr>
<td>8</td>
<td>System overview</td>
<td>53</td>
</tr>
<tr>
<td>9</td>
<td>Distributed context management</td>
<td>55</td>
</tr>
<tr>
<td>10</td>
<td>Context-driven data flows</td>
<td>57</td>
</tr>
<tr>
<td>11</td>
<td>Core concepts</td>
<td>59</td>
</tr>
<tr>
<td>12</td>
<td>Edge programming models</td>
<td>63</td>
</tr>
<tr>
<td>13</td>
<td>APIs and examples of their usage</td>
<td>65</td>
</tr>
<tr>
<td>14</td>
<td>Test</td>
<td>85</td>
</tr>
<tr>
<td>15</td>
<td>Start all core components in the cloud</td>
<td>87</td>
</tr>
<tr>
<td>16</td>
<td>Start an edge node</td>
<td>89</td>
</tr>
<tr>
<td>17</td>
<td>Set up a docker registry for image management</td>
<td>91</td>
</tr>
<tr>
<td>18</td>
<td>Related publications</td>
<td>93</td>
</tr>
<tr>
<td>19</td>
<td>Troubleshooting</td>
<td>95</td>
</tr>
<tr>
<td>20</td>
<td>Contact</td>
<td>97</td>
</tr>
</tbody>
</table>
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 QoS defined by users.

By leveraging these contextual information, 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 huge complexity and 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.

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 a data-centric programming model and a development tool chain for service developers and system integrators to quickly realize IoT services with low development cost and fast time-to-market.

1.1 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.
1.2 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 communication 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.

1.3 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.

1.4 Differentiation

As compared to the other existing IoT edge computing frameworks, such as EdgeX, Azure IoT Edge, Amazon Green-grass, 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 fasion.

More detailed differentiations are summarized in the following table.
Chapter 1. High level introduction

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

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 system. 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.
### 2.1 Start Up FogFlow

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

**Setup FogFlow:**

Download the docker-compose file and the config.json file to setup fogflow.

```
wget https://raw.githubusercontent.com/smartfog/fogflow/master/docker/core/docker-compose.yml
wget https://raw.githubusercontent.com/smartfog/fogflow/master/docker/core/config.json
```

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

- **webportal_ip**: this is the IP address to access the FogFlow web portal provided by Task Designer. It must be accessible from outside by user’s browser.
- **coreservice_ip**: it is used by all edge nodes to access the FogFlow core services, including Discovery, Broker(Cloud), and RabbitMQ;
- **external_hostip**: this is the same as coreservice_ip, for the cloud part of FogFlow;
- **internal_hostip**: is the IP of your default docker bridge, which is the “docker0” network interface on your host

**Important:** please DO NOT use “127.0.0.1” as the IP address of coreservice_ip and external_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, 443, 8080, and 5672 for TCP

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

```
docker-compose pull

docker-compose up -d
```

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

```
root@fffog-yomljrk3y7bs23:~# docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
122a61ece2ce fogflow/master "/master" 26 hours ago Up 26 hours 0.0.0.0.
-0:1060->1060/tcp fogflow_master_1
... e625df79e1e5f1 fogflow/designer "node main.js" 26 hours ago Up 26 hours 0.0.0.0.
-0:80->80/tcp, 0.0.0.0:1030->1030/tcp fogflow_designer_1
```

(continues on next page)
Test the Fogflow Dashboard:
Open the link “http://<webportal_ip>” in your browser to check the status of all FogFlow running components in the cloud. So now you can also check all the components using dashboard.

2.2 Start Up Orion

You may follow the orion docs to set up a Orion Context Broker instance from here: Installing Orion.

You may also setup Orion on docker using below commands.(docker is required this method) Note: Orion container has a dependency on MongoDB database.

Prerequisite: Docker should be installed.

First launch MongoDB container using below command:

```
sudo docker run --name mongodb -d mongo:3.4
```

And then run Orion with this command

```
sudo docker run -d --name orion1 --link mongodb:mongodb -p 1026:1026 fiware/orion --dbhost mongodb
```

Check that everything works with

```
curl http://<Orion IP>:1026/version
```

Note: Allow port 1026 in firewall for public access.

2.3 Program a simple fog function via FogFlow Dashboard

Create a simple Fog Function that:
- accepts “Temperature” Entity Type as SelectCondition, “id” as granularity and “all” as SelectedAttributes,
- publishes a context entity of type “result” in Streams.

2.4 Simulate an IoT device to trigger the Fog Function

There are two ways to trigger the fog function:
1. 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:

```bash
curl -iX POST \\
'http://<Fogflow IP>:8080/ngsi10/updateContext' \\
-H 'Content-Type: application/json' \\
-d ' \\
  { 
    "contextElements": [ 
      { 
        "entityId": { 
          "id": "Temperature001", 
          "type": "Temperature", 
          "isPattern": false 
        },
        "attributes": [ 
          { 
            "name": "temperature", 
            "type": "float", 
            "contextValue": 73 
          } 
        
```
2.4. Simulate an IoT device to trigger the Fog Function
2.5 Check if the fog function is triggered

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

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

2.6 Issue a subscription to forward the generated result to Orion Context Broker

Use the following curl request to subscribe Fogflow Broker to FIWARE Orion:
Please note that this subscription request does not use any restrictions and attributes, it is a general subscription request based of entity type.

### 2.7 Query the result from Orion Context Broker

Visit the following URL in your browser and search for the desired context entities:

```bash
curl http://<Orion IP>:1026/v2/entities/
```
Set up everything on a single machine

To check the basic features of FogFlow, you can set up the entire FogFlow system on a single Linux machine (e.g., Ubuntu 16.04.4 LTS). If you already have Docker-CE and Docker Compose installed on your machine, the setup can be quickly finished in just a few minutes.

Here are the steps to follow:

3.1 Install Docker CE and Docker Compose on your Linux machine

To install Docker CE, please refer to Install Docker CE, required version 18.03.1-ce; please also allow your user to execute the Docker Command without Sudo

To install Docker Compose, please refer to Install Docker Compose, required version 2.4.2

3.2 Download the deployment script and the configuration file

```bash
# download the deployment script
wget https://raw.githubusercontent.com/smartfog/fogflow/master/docker/core/docker-compose.yml

# download the configuration file
wget https://raw.githubusercontent.com/smartfog/fogflow/master/docker/core/config.json
```

3.3 Change the configuration file according to your local environment

You can use the default setting for a simple test, but you need to change the following addresses according to your own environment:
• **webportal_ip**: this is the IP address to access the FogFlow web portal provided by Task Designer. It must be accessible from outside by user’s browser.

• **coreservice_ip**: it is used by all edge nodes to access the FogFlow core services, including Discovery, Broker(Cloud), and RabbitMQ;

• **external_hostip**: this is the same as coreservice_ip, for the cloud part of FogFlow;

• **internal_hostip** is the IP of your default docker bridge, which is the “docker0” network interface on your host.

```json
//you can see the following part in the default configuration file
{
    "webportal_ip": "155.54.239.141",
    "coreservice_ip": "155.54.239.141",
    "external_hostip": "155.54.239.141",
    "internal_hostip": "172.17.0.1",
    ...
}
```

**Important:**

• **firewall rules**: to make your FogFlow web portal accessible via the external_ip; the following ports must be open as well: 80, 443, 8080, and 5672 for TCP

We also assume that you can use the default port numbers for various FogFlow components. More specially, the following ports are required.

• 80: for FogFlow web portal to be accessible at the external IP

• 443: for Discovery to be accessible at the external IP

• 8080: for Broker to be accessible at the external IP

• 5672: for RabbitMQ, used only internally between Master and Worker(s)

### 3.4 Run the downloaded script

```
# pull the docker images of all FogFlow components
docker-compose pull

# start the FogFlow system
docker-compose up -d
```

### 3.5 Test the FogFlow dashboard

Open the link “http://webportal_ip” in your browser to check the status of all FogFlow running components in the cloud.

If everything goes well, you should be able to see the following page from this link.

Furthermore, you should be able to see the status of all core components running in the cloud, from the menu items on the left side of the System Management page.
3.5. Test the FogFlow dashboard

check if you can see the information of the following components started in the cloud:
- IoT Discovery
- IoT Broker (cloud)
- Topology Master
- Worker (cloud)
Chapter 3. Set up everything on a single machine
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 following steps show how to define and test a simple ‘hello world’ function using the web portal provided by FogFlow Task Designer.

4.1 Define a “hello world” fog function

4.1.1 create a fog function from the FogFlow editor

A menu will pop up when you do a right mouse click on the task design board. The displayed menu includes the following items:

- **FogFunction**: to place a fog function element on the design board
- **InputTrigger**: to place a trigger element, which can be linked with a fog function as its input data selector
- **SelectCondition**: to place a filter element, which can be linked with an input trigger in order to specify your input data

Once you click “FogFunction” from the popup menu, a fog function element will be placed on the design board, as shown below.

You can start to configure a fog function once you click the configuration button on the top-right corner, as illustrated by the following figure. Please specify the name of your fog function.
Chapter 4. Define and trigger a fog function
4.1.2 select its input based on entity type

Please click “SelectCondition” from the popup menu to place a “SelectCondition” element on the design board. Configure this element to specify the entity type of your input data. In the following example, we choose “Temperature” as the entity type of input data for the “HelloWorld” fog function.

4.1.3 define a granularity for the creation of its function instances

Please click “InputTrigger” from the popup menu to place a “InputTrigger” element on the design board. Configure it by specifying the following items:

- **SelectedAttributes**: for the selected entity type, which entity attributes are required by your fog function; “all” means to get all entity attributes.

- **Groupby**: this should be one of the selected entity attributes, which defines the granularity of this fog function.

**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 the following example, the granularity is defined by “id”, meaning that FogFlow will create on task instance for each individual entity ID.

**Note**: please link the InputTrigger element to the input of your fog function element; then link your SelectCondition element to the input of your InputTrigger element.

4.1. Define a “hello world” fog function
Chapter 4. Define and trigger a fog function
4.1.4 provide the code of your own function

Currently FogFlow allows developers to specify the function code, either by directly overwriting the following handler function (in Javascript or Python) or by selecting a registered operator.

```javascript
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);
};
```

You can take the example Javascript code above as the implementation of your “HelloWorld” 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.

4.1. Define a “hello world” fog function
Chapter 4. Define and trigger a fog function

1. Customize the implementation of your fog function

```javascript
exports.handler = function(context, entity, publish, query, subscribe) {
    console.log("enter into the user-defined fog function");
    if (contextEntity == null) {
        return;
    } else if (contextEntity.attributes == null) {
        return;
    }
    var updateEntity = {};

    // Code here...
};
```
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

Here are some examples to show how you can use these three call back functions.

• example usage of \texttt{publish}:

```javascript
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 \texttt{query}:

```javascript
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 \texttt{subscribe}:

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

4.1.5 submit your fog function

Once you create the button “Create a Fog Function”, the annotated fog function will be submitted to FogFlow.

4.2 Trigger your “hello world” fog function

The defined “hello world” 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

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

The other way to trigger the “HelloWorld” fog function is to send a NGSI entity update to create the “Temperature” sensor entity. You can run the following command to issue a POST request to the FogFlow broker.

```bash
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", 
          "contextValue": 10 
        } 
      ]
    }
  ]
}
```

(continues on next page)
4.2. Trigger your “hello world” fog function
You can 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

![FogFlow screenshot](image_url)

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

![FogFlow screenshot](image_url)
4.2. Trigger your “hello world” fog function
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 “requirement” object.

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

5.1 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.
Chapter 5. Define and trigger a service topology
5.2 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

5.3 Register your task operators

For each operator, once we create its docker image and push it to the FogFlow docker registry, we must register the operator in FogFlow. This 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.

5.3.1 Register it via FogFlow Task Designer

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

![Image of list of docker images](image)

After clicking the “register” button, you can see a form 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 you used to publish your operator docker image; by default it is “latest”
- Hardware Type: the hardware type that your docker image supports, including X86 or ARM (e.g. Raspberry Pi)
• OS Type: the operating system type that your 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 your docker image must be consistent with the one you publish to Docker Hub. By default, FogFlow will fetch the required docker images from Docker Hub using the name you register here for your operator.

5.3.2 Register it programmatically by sending a NGSI update

You can also register an operator docker image by sending 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 docker image.

**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
```
-d ' 
{  
  "contextElements": [  
   {  
    "entityId": {  
      "id": "counter:latest",  
      "type": "DockerImage",  
      "isPattern": false  
    },  
    "attributes": [  
     {  
      "name": "image",  
      "type": "string",  
      "contextValue": "counter"  
     },  
     {  
      "name": "tag",  
      "type": "string",  
      "contextValue": "latest"  
     },  
     {  
      "name": "hwType",  
      "type": "string",  
      "contextValue": "X86"  
     },  
     {  
      "name": "osType",  
      "type": "string",  
      "contextValue": "Linux"  
     },  
     {  
      "name": "operatorName",  
      "type": "string",  
      "contextValue": "counter"  
     },  
     {  
      "name": "prefetched",  
      "type": "string",  
      "contextValue": false  
     }],  
    "domainMetadata": [  
     {  
      "name": "operator",  
      "type": "string",  
      "value": "counter"  
     }  
    ]  
  },  
  "updateAction": "UPDATE" 
}
```

JavaScript

```
var image = {  
    name: "counter",  
    tag: "latest",  
    hwType: "X86", 
};
```

(continues on next page)
```javascript
osType: "Linux",
operatorName: "counter",
prefetched: false
};

// register a new docker image
var 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.operatorName};
newImageObject.attributes.prefetched = {type: 'boolean', value: image.prefetched};

newImageObject.metadata = {};
newImageObject.metadata.operator = {
  type: 'string',
  value: image.operatorName
};

// assume the config.brokerURL is the IP of cloud IoT Broker
var client = new NGSI10Client(config.brokerURL);
client.updateContext(newImageObject).then(function(data) {
  console.log(data);
}).catch(function(error) {
  console.log('failed to register the new device object');
});
```

### 5.4 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.

#### 5.4.1 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
   - priority: define the priority level of all tasks in your topology, which will be utilized by edge nodes to decide how resource should be assigned to tasks

---

Chapter 5. Define and trigger a service topology
• 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

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 to define your data processing flows according to the design you had in mind.

3. **define Task Profile for each task in the data flow, including** As shown in the following picture, you can start to specify the profile of each task 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
   
   • 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 use “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

   • shuffling of input streams: to indicate how the input stream should be assigned to the instance(s) of this task during the runtime, including the following two cases
      
      – “shuffling” = “broadcast”: the selected input streams should be repeatedly assigned to every task instance of this operator

5.4. Specify a service topology
- “shuffling” = “unicast”: each of the selected input streams should be assigned to a specific task instance only once
  
• entity type of output streams: to specify the entity type of the produced output stream

5.4.2 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 examples to register an operator docker image.

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

curl -iX POST \
'http://localhost:8070/ngsi10/updateContext' \
-H 'Content-Type: application/json' \
-d '  
  "contextElements": [  
    {  
      "entityId": {  
        "id": "Topology.anomaly-detection",  
        "type": "anomaly-detection",  
        "isPattern": false  
      },  
      "attributes": [  
        {  
          "name": "status",  
          "type": "string",  
          "contextValue": "enabled"  
        },  
        {  
          "name": "template",  
          "type": "object",  
          "contextValue": {  
            "description": "detect anomaly events from time series data points",  
            "name": "anomaly-detection",  
            "priority": {  
              "exclusive": false,  
              "trigger": "on-demand",  
              "tasks": [  
                {  
                  "name": "AnomalyDetector",  
                  "operator": "anomaly",  
                  "groupBy": "shop",  
                  "input_streams": []  
                }  
              ]  
            }  
          }  
        }  
      ]  
    }  
  ]'
JavaScript

```javascript
// 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 = {
    "description": "detect anomaly events from time series data points",
    "name": "anomaly-detection",
    "priority": {
        "exclusive": false,
        "level": 100
    },
    "trigger": "on-demand",
    "tasks": [
        {
            "type": "PowerPanel",
            "shuffling": "unicast",
            "scoped": true
        },
        {
            "type": "Rule",
            "shuffling": "broadcast",
            "scoped": false
        }
    ],
    "output_streams": [
        {
            "type": "Anomaly"
        }
    ],
    "name": "Counter",
    "operator": "counter",
    "groupBy": "*",
    "input_streams": [
        {
            "type": "Anomaly",
            "shuffling": "unicast",
            "scoped": true
        }
    ],
    "output_streams": [
        {
            "type": "Stat"
        }
    ]
}
```

5.4. Specify a service topology
"name":"AnomalyDetector",
"operator":"anomaly",
"groupBy":"shop",
"input_streams":[
  
  {
    "type": "PowerPanel",
    "shuffling": "unicast",
    "scoped": true
  },
  {
    "type": "Rule",
    "shuffling": "broadcast",
    "scoped": false
  }
],
"output_streams":[
  
  ]
},

{ "name": "Counter",
"operator": "counter",
"groupBy": "*",
"input_streams": [
  {
    "type": "Anomaly",
    "shuffling": "unicast",
    "scoped": true
  }
],
"output_streams": [
  {
    "type": "Stat"
  }
]
},

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

// 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');
});

5.5 Trigger the service topology by sending a customized requirement

Once developers submit a specified service topology and the implemented operators, the service data processing logic can be triggered on demand by a high level processing requirement. The processing requirement is sent as NGSI10 update, with the following properties:

- topology: which topology to trigger
- expected output: the output stream type expected by external subscribers
- scope: a defined geoscope for the area where input streams should be selected
- scheduler: which type of scheduling method should be chosen by Topology Master for task assignment

**Important:** Please notice that the input data for the leaf nodes of your service topology must be already reported by your IoT devices. It works like this in the current version, but this requirement will be no longer needed in the new version.

Also, the entity ID of your input data must follow a pattern: “Stream.[Type].[xxx]”. For example, for a temperature sensor with “Temperature” as the entity type, the entity ID of this temperature entity requires to be “Stream.Temperature.[xxx]”, for example, “Stream.Temperature.001”. Please look at the example in the section

Here are the Curl and the Javascript-based code examples to trigger a service topology by sending a customized requirement entity to FogFlow.

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

curl

```bash
curl -iX POST \
  'http://localhost:8070/ngsi10/updateContext' \
-H 'Content-Type: application/json' \
-d '{
  "contextElements": [
  {
    "entityId": {
      "id": "Requirement.163ed933-828c-4c20-ab2a-59f73e8682cf",
      "type": "Requirement",
      "isPattern": false
    },
    "attributes": ["output"
  ]
}
'}
```


```javascript
var rid = 'Requirement.' + uuid();

var requirementCtxObj = {
  entityId: {
    id: rid,
    type: 'Requirement',
    isPattern: false
  };

var restriction = {
  scopes: [{
    scopeType: geoscope.type,
    scopeValue: geoscope.value
  }];

requirementCtxObj.attributes = {
  output: {
    type: 'string',
    value: 'Stat'
  },
  scheduler: {
    type: 'string',
    value: 'closest_first'
  },
  restriction: {
    type: 'object',
    value: restriction
  }
};

requirementCtxObj.metadata = {
  topology: {
    type: 'string',
    value: curTopology.entityId.id
  }
};

console.log(requirementCtxObj);
```

(continues on next page)
5.5. Trigger the service topology by sending a customized requirement

```javascript
// assume the config.brokerURL is the IP of cloud IoT Broker
var client = new NGSI10Client(config.brokerURL);
client.updateContext(requirementCtxObj).then( function(data) {
    console.log(data);
}).catch( function(error) {
    console.log('failed to send a requirement');
});
```
Connect an IoT device to FogFlow

If your device can communicate with FogFlow via NGSI, connecting your device to FogFlow can be very easy. It requires some small application to be running on your device, for example, a raspberry Pi with several connected sensors or actuators.

In the following example, we show how a simulated PowerPanel device can be connected to FogFlow via NGSI. This example code is also accessible from FogFlow code repository in the application folder.

You need Node.js to run this example code. Please install Node.js and npm.

```javascript
'use strict';

const NGSI = require('./ngsi/ngsiclient.js');
const fs = require('fs');

// read device profile from the configuration file
var args = process.argv.slice(2);
if(args.length != 1){
    console.log('please specify the device profile');
    return;
}

var cfgFile = args[0];
var profile = JSON.parse(fs.readFileSync(cfgFile));

var ngsi10client;
var timer;

// find out the nearby IoT Broker according to my location
var discovery = new NGSI.NGSI9Client(profile.discoveryURL);
discovery.findNearbyIoTBroker(profile.location, 1).then(
    function(brokers) {
        console.log('--------nearbybroker---------');
        console.log(brokers);
        console.log('-------------end-------------');
    });
```

(continues on next page)
if (brokers && brokers.length > 0) {
  ngsi10client = new NGSI.NGSI10Client(brokers[0]);

  // generating data observations periodically
  timer = setInterval(function()
    updateContext();
    }, 1000);

  // register my device profile by sending a device update
  registerDevice();
})
});

// register device with its device profile
function registerDevice() {
  var ctxObj = {};
  ctxObj.entityId = {
    id: 'Device.' + profile.type + '.' + profile.id,
    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');
  });
}

// update context for streams
function updateContext() {

(continues on next page)
```javascript
var ctxObj = {};
ctxObj.entityId = {
  id: 'Stream.' + profile.type + '.' + profile.id,
  type: profile.type,
  isPattern: false
};
ctxObj.attributes = {};

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

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

process.on('SIGINT', function() {
  if(ngsi10client) {
    clearInterval(timer);
    // to delete the device
    var entity = {
      id: 'Device.' + profile.type + '.' + profile.id,
      type: 'Device',
      isPattern: false
    };
    ngsi10client.deleteContext(entity).then(function(data) {
      console.log(data);
    }).catch(function(error) {
      console.log('failed to delete context');
    });
    // to delete the stream
    var entity = {
```
id: 'Stream.' + profile.type + '.' + profile.id,
type: 'Stream',
isPattern: false
};
ngsi10client.deleteContext(entity).then(function(data) {
    console.log(data);
}).catch(function(error) {
    console.log('failed to delete context');
});
}
});

You need to modify discoveryURL in profile1.json.

```
{
    "discoveryURL": "http://35.198.104.115:443/ngsi9",
    "location": {
        "latitude": 35.692221,
        "longitude": 139.709059
    },
    "iconURL": "/img/shop.png",
    "type": "PowerPanel",
    "id": "01"
}
```

You need to install the packages as follows:

```
npm install
```

Run this example code as follows:

```
node powerpanel.js profile1.json
```
Since November 2017 FogFlow has been promoted as an incubated open source Generic Enabler (GE) in the FIWARE community. Within this community, FogFlow holds a unique position as Cloud-Edge Orchestrator to launch and manage dynamic data processing flows seamlessly over cloud and edges for data ingestion, transformation, and also advanced analytics.

As illustrated by the following picture, FogFlow can interwork with other FIWARE GEs to power FIWARE-based IoT Platforms at the following two layers. At the upper layer, FogFlow can interwork with Orion Broker via the standardized NGSI interface in the following two ways.

### 7.1 Orion Broker as the Destination

The first way is to consider Orion Broker as the destination of any context information generated by a FogFlow IoT service. In this case a NGSI subscription must be issued by an external application or FogFlow Dashboard to ask FogFlow to forward the requested context updates to a specified Orion Broker.

Two ways are provided to tell FogFlow which entitie should be forwarded to the Orion Broker. The first way is to issue a raw subscription to FogFlow Broker. The second way is to write a small JavaScript program to do this. Examples are provided below. Notice that the integration is using the NGSI V2 interface of Orion Broker.

**Important:**

- **fogflowBroker**: IP address of the FogFlow Broker, it could be “webportal_ip” or “coreservice_ip” in the configuration file. This is up to where you access the FogFlow system.
- **orionBroker**: the accessible IP address of your Orion Running Instance.

```bash
curl -iX POST \\ 'http://fogflowBroker:8080/ngsi10/subscribeContext' \\ -H 'Content-Type: application/json'
```

(continues on next page)
-H 'Destination: orion-broker' \
-d '{"entities": [{"type": "PowerPanel", "isPattern": true}, 
    "reference": "http://orionBroker:1026/v2"}]'

JavaScript

```javascript

// entityType: the type of context entities to be pushed to Orion Broker
// orionBroker: the URL of your running Orion Broker
function subscribeFogFlow(entityType, orionBroker) {
  var subscribeCtxReq = {};
  subscribeCtxReq.entities = [{type: entityType, isPattern: true}];
  subscribeCtxReq.reference = 'http://' + orionBroker + '/v2';

  client.subscribeContext4Orion(subscribeCtxReq).then(function(subscriptionId) {
    console.log(subscriptionId);
    ngsipropxy.reportSubID(subscriptionId);
  }).catch(function(error) {
    console.log('failed to subscribe context');
  });
}

// client to interact with IoT Broker
var client = new NGSI10Client(config.brokerURL);
subscribeFogFlow('PowerPanel', 'cpaasio-fogflow.inf.um.es:1026');
```

To verify if the requested data has been pushed to your Orion Broker, you can send the following NGSI v2 query to check it.
7.2 Orion Broker as a Data Source

The second way is to consider Orion Broker as a data source to provide additional information. In this case we can implement a simple fog function to fetch any necessary information into the FogFlow system. In either way there is no need to make any change to the existing Orion-based FIWARE system. Therefore, this type of integration can be done fast with nearly zero effort. At the low layer, for the integration with any Non-NGSI supported devices like MQTT, COAP, OneM2M, OPC-UA, LoRaWAN, FogFlow can reuse the modules of existing IoT agents and transform them into FogFlow adapters based on the fog function programming model. With these adapters FogFlow can dynamically launch necessary adapters for device integration directly at edges. This way FogFlow is able to talk with a wide range of IoT devices.

Currently, the MQTT based adapter has been provided and it is also open source in a separated repository. Please refer to https://github.com/smartfog/adapter

In the future, we consider to provide an adapter to IoT-agents. As illustrated by the following figure, this adapter could be reused to integrate with all types of FIWARE IoT agents.
8.1 System architecture

The FogFlow framework operates on a geo-distributed, hierarchical, and heterogeneous ICT infrastructure that includes cloud nodes, edge nodes, and IoT devices. The following figure illustrates the system architecture of FogFlow and its major components across three logical layers.
8.2 Three layers

Logically, FogFlow consists of the following three layers:

- **service management**: convert service requirements into concrete execution plan and then deploy the generated execution plan over cloud and edges
- **context management**: manage all context information and make them discoverable and accessible via flexible query and subscribe interfaces
- **data processing**: launch data processing tasks and establish data flows between tasks via the pub/sub interfaces provided by the context management layer

8.3 System components

centralized service components to be deployed in the cloud

- **Task Designer**: provide the web-based interface for service providers to specify, register, and manage their tasks and service topologies;
- **Topology Master**: figure out when and which task should be instantiated, dynamically configure them, and also decide where to deploy them over cloud and edges;
- **IoT Discovery**: manage all registered context availability information, including its ID, entity type, attribute list, and metadata; allow other components to query and subscribe their interested context availability information via NGSI9

distributed components to be deployed both in the cloud and at edges

- **Worker**: according to the assignment from the topology master, each worker will launch its scheduled task instances in docker containers on its local host; configure their inputs and outputs and manage all task instances locally based on task priority
- **IoT Broker**: each broker manages a part of context entities published by nearby IoT devices and also provides a single view of all context entities for IoT devices to query and subscribe the entities they need

external service components to be used in FogFlow

- **Dock Registry**: managed all docker images provided by developers;
- **RabbitMQ**: The internal communication between topology master and the workers
- **PostgreSQL**: the backend database to save the registered context availability information
The context management system is designed to provide a global view for all system components and running task instances to query, subscribe, and update context entities via the unified data model and the communication protocol NGSI. It plays a very important role to support the standard-based edge programming model in FogFlow. As compared to other existing brokers like MQTT-based Mosquitto or Apache Kafka, the distributed context management system in FogFlow has the following features:

- separating context availability and context entity
- providing separated and standardized interfaces to manage both context data (via NGSI10) and context availability (via NGSI9).
- supporting not only ID-based and topic-based query and subscription but also geoscope-based query and subscription

As illustrated by the following figure, in FogFlow a large number of distributed IoT Brokers work in parallel under the coordination a global and centralized IoT Discovery.
9.1 IoT Discovery

The centralized IoT Discovery provides a global view of context availability of context data and provides NGSI9 interfaces for registration, discovery, and subscription of context availability.

9.2 IoT Broker

The IoT Broker in Fogflow is very light-weight, because it keeps only the lastest value of each context entity and saves each entity data directly in the system memory. This brings high throughput and low latency for the data transfer from context producers to context consumers.

Each IoT Broker manages a portion of the context data and registers data to the shared IoT Discovery. However, all IoT Brokers can equally provide any requested context entity via NGSI10 because they can find out which IoT Broker provides the entity through the shared IoT Discovery and then fetch the entity from that remote IoT Broker.
In FogFlow each worker is an agent to perform the following two type of deployment actions assigned by Topology Master.

- start/terminate a task instance
- add/remove an input stream to an existing running task instance

For example, each worker takes the following steps to start a task instance.

1. the worker asks its local docker engine to create a new container for running the new task instance with a specified listening port number
2. the task instance starts to run and listening on the given port number to receive input data streams
3. the worker sends a configuration object to the running task instance via its listening port
4. the worker issues NGSI10 subscriptions on behalf of the running task instance
5. the required input data streams flow into the running task instance for further processing via its listening port
6. the running task instance processes the received input stream data and then generate its result
7. the running task instance publishes the generated result as its output by sending NGSI10 updates
Chapter 10. Context-driven data flows
Core concepts

11.1 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.

11.2 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`.

## 11.3 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.

## 11.4 Service Template

Each IoT service is described by a service template, which can be a service topology with a set of linked operators or a fog function with a single operator. For example, when we use a service topology to specify your service template, the following information will be included.

- **topology name**: the unique name of your topology
- **service description**: some text to describe what this service is about
- **priority**: define the priority level of all tasks in your 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.

## 11.5 Dynamic data flow

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

12.1 Service Topology

The first workload pattern is to trigger necessary processing flows to produce some output data only when the output data are requested by consumers. To define an IoT service based on this pattern, the service provider needs to define a service topology, which consists of a set of linked operators and each operator is annotated with a specific granularity. The granularity of an operator will be taken into account by FogFlow to decide how many task instances of such an operator should be instantiated based on the available data.

A service topology must be triggered explicitly by a requirement object issued by a consumer or any application. The requirement object defines which part of processing logic in the defined service topology needs to be triggered and it can also optionally define a specific geo-scope to filter out data sources for applying the triggered processing logic.

12.2 Fog Function

The second workload pattern is designed for the scenario in which service designers do not a-priori know the exact sequence of stream processing steps. Instead they can define a fog function to include a specific operator for handling a given type of information. FogFlow can then create the graph of processing flows based on this description of all fog functions. Different from service topology, a fog function is a very simple topology with only one operator and it is triggered when its input data become available. As FogFlow can automatically chain different fog functions as well as allow more than one fog functions to handle a new data item, a constantly changing execution graph can be automatically triggered and managed by the FogFlow runtime as data arrive and disappear. From the design perspective, fog function is more flexible than service topology, because the overall processing logic of an IoT service can be easily changed over time by adding or removing fog functions when the service processing logic needs to modify for new business requirements. With the fog function programming model, FogFlow can support serverless computing for a cloud-edge based environment.
Chapter 12. Edge programming models
13.1 APIs of FogFlow Discovery

13.1.1 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**

<table>
<thead>
<tr>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>latitude</td>
<td>latitude of your location</td>
</tr>
<tr>
<td>longitude</td>
<td>latitude of your location</td>
</tr>
<tr>
<td>limit</td>
<td>number of expected brokers</td>
</tr>
</tbody>
</table>

Please check the following examples.

**Note:** For the Javascript code example, you need the library ngsiclient.js. Please refer to the code repository at application/device/powerpanel

```bash
curl -iX POST \\
'http://localhost:8071/ngsi9/discoverContextAvailability' \\
-H 'Content-Type: application/json' \\
-d '{
    "entities": [
        {
            "type": "IoTBroker",
            "isPattern": true
        }
    ]
}'
```

(continues on next page)
JavaScript

```javascript
const NGSI = require('./ngsi/ngsiclient.js');

var discoveryURL = "http://localhost:8071/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);
    });
```

13.2 APIs of FogFlow Broker

13.2.1 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

<table>
<thead>
<tr>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>latitude</td>
<td>latitude of your location</td>
</tr>
<tr>
<td>longitude</td>
<td>latitude of your location</td>
</tr>
<tr>
<td>limit</td>
<td>number of expected brokers</td>
</tr>
</tbody>
</table>
Example:

curl

curl -iX POST \\
'http://localhost:8070/ngsi10/updateContext' \\
-H 'Content-Type: application/json' \\
-d ' \\
  \\
  "contextElements": [ \\
    { \\
      "entityId": { \\
        "id": "Device.temp001", \\
        "type": "Temperature", \\
        "isPattern": false \\
      }, \\
      "attributes": [ \\
        { \\
          "name": "temp", \\
          "type": "integer", \\
          "contextValue": 10 \\
        } \\
      ], \\
      "domainMetadata": [ \\
        { \\
          "name": "location", \\
          "type": "point", \\
          "value": { \\
            "latitude": 49.406393, \\
            "longitude": 8.684208 \\
          } \\
        }, \\
        { \\
          "name": "city", \\
          "type": "string", \\
          "value": "Heidelberg" \\
        } \\
      ] \\
    }, \\
    \\
    "updateAction": "UPDATE" \\
  ]' \\

JavaScript

```javascript
const NGSI = require('./ngsi/ngsiclient.js');
var brokerURL = "http://localhost:8070/ngsi10"

var ngsi10client = new NGSI.NGSI10Client(brokerURL);

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

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

(continues on next page)
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');
});

13.2.2 Query Context via GET

Fetch a context entity by ID

GET /ngsi10/entity/#eid

<table>
<thead>
<tr>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eid</td>
<td>entity ID</td>
</tr>
</tbody>
</table>

Example:

curl http://localhost:8070/ngsi10/entity/Device.temp001

Fetch a specific attribute of a specific context entity

GET /ngsi10/entity/#eid/#attr

<table>
<thead>
<tr>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eid</td>
<td>entity ID</td>
</tr>
<tr>
<td>attr</td>
<td>specify the attribute name to be fetched</td>
</tr>
</tbody>
</table>
Example:

```
curl http://localhost:8070/ngsi10/entity/Device.temp001/temp
```

Check all context entities on a single Broker

GET /ngsi10/entity

Example:

```
curl http://localhost:8070/ngsi10/entity
```

13.2.3 Query context via POST

POST /ngsi10/queryContext

<table>
<thead>
<tr>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>entityId</td>
<td>specify the entity filter, which can define a specific entity ID, ID pattern, or type</td>
</tr>
<tr>
<td>restriction</td>
<td>a list of scopes and each scope defines a filter based on domain metadata</td>
</tr>
</tbody>
</table>

query context by the pattern of entity ID

```
curl -X POST 'http://localhost:8070/ngsi10/queryContext' \ 
  -H 'Content-Type: application/json' \ 
  -d '{"entities": [{"id":"Device.*","isPattern":true}]}'
```

JavaScript

```javascript
const NGSI = require('./ngsi/ngsiclient.js');
var brokerURL = "http://localhost:8070/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 -X POST 'http://localhost:8070/ngsi10/queryContext' \ 
  -H 'Content-Type: application/json' \ 
  -d '{"entities": [{"type":"Temperature","isPattern":true}]}'
```
JavaScript

```javascript
const NGSI = require('./ngsi/ngsiclient.js');
var brokerURL = "http://localhost:8070/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

curl -X POST 'http://localhost:8070/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
            }
        }]
    }
}'

JavaScript

```javascript
const NGSI = require('./ngsi/ngsiclient.js');
var brokerURL = "http://localhost:8070/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
    }
}]};
```

(continues on next page)
query context by geo-scope (polygon)

curl

curl -X POST 'http://localhost:8070/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
            },
            {
              "latitude": 34.4069096565206,
              "longitude": 141.51489257812503
            },
            {
              "latitude": 34.4069096565206,
              "longitude": 135.84594726562503
            }
          ]
        }
      }
    ]
  }
'}
const NGSI = require('./ngsi/ngsiclient.js');
var brokerURL = "http://localhost:8070/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}
            ]
        }
    ]
}

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

curl -X POST 'http://localhost:8070/ngsi10/queryContext' \
-H 'Content-Type: application/json' \
(continues on next page)
JavaScript

```javascript
const NGSI = require('./ngsi/ngsiclient.js');
var brokerURL = "http://localhost:8070/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

```bash
curl -X POST 'http://localhost:8070/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"
    }]
  }
}
```

(continues on next page)
13.2.4 Delete context

Delete a specific context entity by ID

DELETE /ngsi10/entity/#eid

<table>
<thead>
<tr>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eid</td>
<td>entity ID</td>
</tr>
</tbody>
</table>

Example:

curl -iX DELETE http://localhost:8070/ngsi10/entity/Device.temp001

13.2.5 Subscribe context

POST /ngsi10/subscribeContext

<table>
<thead>
<tr>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>entityId</td>
<td>specify the entity filter, which can define a specific entity ID, ID pattern, or type</td>
</tr>
<tr>
<td>restriction</td>
<td>a list of scopes and each scope defines a filter based on domain metadata</td>
</tr>
<tr>
<td>reference</td>
<td>the destination to receive notifications</td>
</tr>
</tbody>
</table>
subscribe context by the pattern of entity ID

curl

curl -X POST 'http://localhost:8070/ngsi10/subscribeContext' \
-H 'Content-Type: application/json' \
-d '{
   "entities": ["Device.*","isPattern":true],
   "reference": "http://localhost:8066"
}'

JavaScript

```javascript
const NGSI = require('./ngsi/ngsiclient.js');
var brokerURL = "http://localhost:8070/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

curl -X POST 'http://localhost:8070/ngsi10/subscribeContext' \
-H 'Content-Type: application/json' \
-d '{
   "entities": ["Temperature","isPattern":true]
   "reference": "http://localhost:8066"
}'

JavaScript

```javascript
const NGSI = require('./ngsi/ngsiclient.js');
var brokerURL = "http://localhost:8070/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

curl -X POST 'http://localhost:8070/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
            }
        }]
    }
}
}

JavaScript

```javascript
const NGSI = require('./ngsi/ngsiclient.js');
var brokerURL = "http://localhost:8070/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
curl -X POST 'http://localhost:8070/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:8070/ngsi10"
var ngsi10client = new NGSI.NGSI10Client(brokerURL);

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 -X POST 'http://localhost:8070/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
      }
    }]
  }
}';
```
JavaScript

```javascript
const NGSI = require('./ngsi/ngsiclient.js');
var brokerURL = "http://localhost:8070/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
  }
}, {
  "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

<table>
<thead>
<tr>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>the subscription ID created when the subscription is issued</td>
</tr>
</tbody>
</table>

curl -iX DELETE http://localhost:8070/ngsi10/subscription/#sid

### 13.3 APIs of FogFlow Service Orchestrator

The overall development process of an IoT Service in FogFlow is shown in the following figure. For the development of a fog function, the steps 4 and 5 are combined, which means a default requirement is issued by the FogFlow editor when a fog function is submitted.
13.3.1 Implement an operator

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.

- nodejs-based
- python-based

**Note:** currently two templates are provided: one for nodejs based Implement and the other for python-based implementation

13.3.2 Publish the operator

You can publish the image of your operator to the public docker registry or your own private docker registry. If you do not want to use any docker registry, you have to make sure that the docker image of your operator is built on all edge nodes. Currently, when the FogFlow worker receives a command to launch a task instance, it will first search the required docker image from the local storage. If it does not find it, it will start to fetch the required docker image for the docker registry (the public one or your own private one, which is up to the configuration of your FogFlow worker).

If you like to publish your image, you can use the following docker command.

```
docker push [the name of your image]
```

**Note:** this step is done with only docker commands

13.3.3 Define and register your operator

You can also register an operator docker image by sending a constructed NGSI update message to the IoT Broker deployed in the cloud.

Here is a Javascript-based code example to register an operator docker image. Within this 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.

```javascript
var image = {
    name: "counter",
    tag: "latest",
    hwType: "X86",
    osType: "Linux",
    operatorName: "counter",
    prefetched: false
};

// register a new docker image
var newImageObject = {};

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

(continues on next page)
13.3.4 Define and register your service topology

Usually, you can define and register your service topology via the FogFlow topology editor. However, you can also define and register with your own code.

To register a service topology, your code needs to send a constructed NGSI update message to the IoT Broker deployed in the cloud.

Here is a Javascript-based code example to register an operator docker image. Within this 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.

```
// 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 = {
    "description": "detect anomaly events from time series data points",
    "name": "anomaly-detection",
    "priority": {
        "exclusive": false,
        "level": 100
    },
    "trigger": "on-demand",
    "tasks": [
        {
            "name": "AnomalyDetector",
            "operator": "anomaly",
            "groupBy": "shop",
            "input_streams": [
                {
                    "type": "PowerPanel",
                    "shuffling": "unicast",
                    "scoped": true
                }
            ]
        }
    ]
};
```
//submit it to FogFlow via NGSI Update
var topologyCtxObj = {};

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

topologyCtxObj.attributes = {};
topologyCtxObj.attributes.status = {type: 'string', value: 'enabled'};
topologyCtxObj.attributes.template = {type: 'object', value: 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');
});
13.3.5 Create a requirement entity to trigger the service topology

Here is the Javascript-based code example to trigger a service topology by sending a customized requirement entity to FogFlow.

```javascript
var rid = 'Requirement.' + uuid();

var requirementCtxObj = { id : rid,
    type: 'Requirement',
    isPattern: false
};

var restriction = { scopes:[{scopeType: geoscope.type, scopeValue: geoscope.value}]};

requirementCtxObj.attributes = {};
requirementCtxObj.attributes.output = {type: 'string', value: 'Stat'};
requirementCtxObj.attributes.scheduler = {type: 'string', value: 'closest_first'};
requirementCtxObj.metadata = {};
requirementCtxObj.metadata.topology = {type: 'string', value: curTopology.entityId.id}

console.log(requirementCtxObj);

// assume the config.brokerURL is the IP of cloud IoT Broker
var client = new NGSI10Client(config.brokerURL);
client.updateContext(requirementCtxObj).then( function(data) {
    console.log(data);
}).catch( function(error) {
    console.log('failed to send a requirement');
});
```

13.3.6 Remove a requirement entity to terminate the service topology

Here is the Javascript-based code example to terminate a service topology by deleting the requirement entity.

```javascript
var rid = [the id of your created requirement entity];

//
var client = new NGSI10Client(config.brokerURL);
client.deleteContext(rid).then( function(data) {
    console.log(data);
}).catch( function(error) {
    console.log('failed to send a requirement');
});
```
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
Start all core components in the cloud

The steps are the same as you set up everything on a single machine, see Steps.
Chapter 15. Start all core components in the cloud
Typically, an FogFlow edge node needs to deploy a Worker and an IoT broker. 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:

### 16.1 Install Docker Engine

To install Docker CE, please refer to [install_docker], required version 18.03.1-ce. *please also allow your user to execute the Docker Command without Sudo*

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

### 16.2 Download the deployment script

```bash
#download the deployment scripts
wget https://raw.githubusercontent.com/smartfog/fogflow/master/docker/edge/start.sh
wget https://raw.githubusercontent.com/smartfog/fogflow/master/docker/edge/stop.sh

#make them executable
chmod +x start.sh stop.sh
```
16.3 Download the default configuration file

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

16.4 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;
- **internal_hostip** is the IP of your default docker bridge, which is the “docker0” network interface on your host.

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

16.5 Start both Edge IoT Broker and FogFlow Worker

**Note:** if the edge node is ARM-based, please attach arm as the command parameter

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

# if the edge node is ARM-based, please attach arm as the command parameter
#.start.sh arm
```

16.6 Stop both Edge IoT Broker and FogFlow Worker

```bash
# stop both components in the same script
./stop.sh
```
CHAPTER 17

Set up a docker registry for image management

17.1 Create a Self Signed Certificate

You need to create a self signed certificate on your server to use it for the private Docker Registry.

```bash
mkdir registry_certs
openssl req -newkey rsa:4096 -nodes -sha256 -keyout registry_certs/domain.key -x509 -days 356 -out registry_certs/domain.cert
ls registry_certs/
```

Finally you have two files:

- `domain.cert` – this file can be handled to the client using the private registry
- `domain.key` – this is the private key which is necessary to run the private registry with TLS

17.2 Run the Private Docker Registry with TLS

Now we can start the registry with the local domain certificate and key file:

```bash
docker run -d -p 5000:5000 -v $(pwd)/registry_certs:/certs -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.cert -e REGISTRY_HTTP_TLS_KEY=/certs/domain.key --restart=always --name registry registry:2
```

Here we map the folder `/registry_certs` as an volume into the docker registry container. We use environment variables pointing to the certificate and key file.

Now you can push your local image into the new registry:
17.3 Access the Remote Registry form a fog node

Now as the private registry is started with TLS Support you can access the registry from any client which has the domain certificate. Therefore the certificate file “domain.cert” must be located on the client in a file

/etc/docker/certs.d/<registry_address>/ca.cert

Where <registry_address> is the server host name. After the certificate was updated you need to restart the local docker daemon:

mkdir -p /etc/docker/certs.d/dock01:5000
cp domain.cert /etc/docker/certs.d/dock01:5000/ca.crt
service docker restart

Now finally you can push you images into the new private registry:

docker tag imixs/proxy dock01:5000/proxy:dock01
docker push dock01:5000/proxy:dock01

17.4 Start Docker Registry Frontend

The project konradkleine/docker-registry-frontend provides a cool web front-end which can be used to simplify access to the registry through a web browser. The docker-registry-frontend can be started as a docker container. Assuming your registry runs on

https://yourserver.com:5000

use the following docker run command to start the frontend container:

docker run \
-d \
  -e ENV_DOCKER_REGISTRY_HOST=yourserver.com \
  -e ENV_DOCKER_REGISTRY_PORT=5000 \
  -e ENV_DOCKER_REGISTRY_USE_SSL=1 \
  -p 0.0.0.0:80:80 \
  konradkleine/docker-registry-frontend:v2

You can now access your registry via web browser url:

http://localhost:80/
Related publications


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: 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.
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.