

ProximaSafe: Joining the Dots in OCI to build a Stream Analysis Lab

Chapter Two: From Edge to Serverless

*I set a course just east of Lyra
And northwest of Pegasus
Flew into the light of Deneb
Sailed across the Milky Way
(Neil Peart, 1977)*

A quick recap

Greetings, and welcome to **Chapter Two** of our journey!

In the *previous article* we showed the ProximaSafe scope, overall architecture and the components need to achieve our goal: get the stream flow coming from a determined edge environment in OCI, perform the analysis of the stream to detect possible anomalies and send back the errors to the edge in order to carry out corrective actions. All this with development boards commercially available (almost) anywhere and easy to pack and transport anywhere.

Now it is time to having fun fiddling with sensors and OCI Functions, covering a number of areas such as:

- **Selecting** components our MicroEdge lab environment.
- **Identifying** functionalities and libraries for Edge components, both emitters and receivers (publishers and subscribers).
- **Configuring and Bridging** our local MQTT server with the Cloud MQTT server residing on OCI Compute.
- **Develop** a Serverless component in [Oracle Functions](#) to return error conditions and warnings to the edge.
- **Creating** an API deployment addressing the serverless function, to be accessed from the Stream Analytics module described in Chapter One.

That said, without further ado let's dive into some practical aspects of the matter.

Selecting the edge components

During the spring of 2020 (and the relating lockdown) I fell - almost immediately - in love with the [M5Stack](#) development boards series, based on the [ESP32 microcontroller](#). These cute little boxes have an integrated display, which - sometimes - is useful to help with building simple and intuitive on-board GUIs (that's not my case, I'll always be an ASCII fanboy) or debugging and showing messages contents without bothering to open a serial terminal from the Arduino IDE. Furthermore, a sumptuous choice of different programming models, IDEs and languages is available:

- [Arduino](#) mode and the related [IDE](#): the elected programming language is **C/C++**, it is fun and suitable for almost-extinct IT apatosaurus like myself

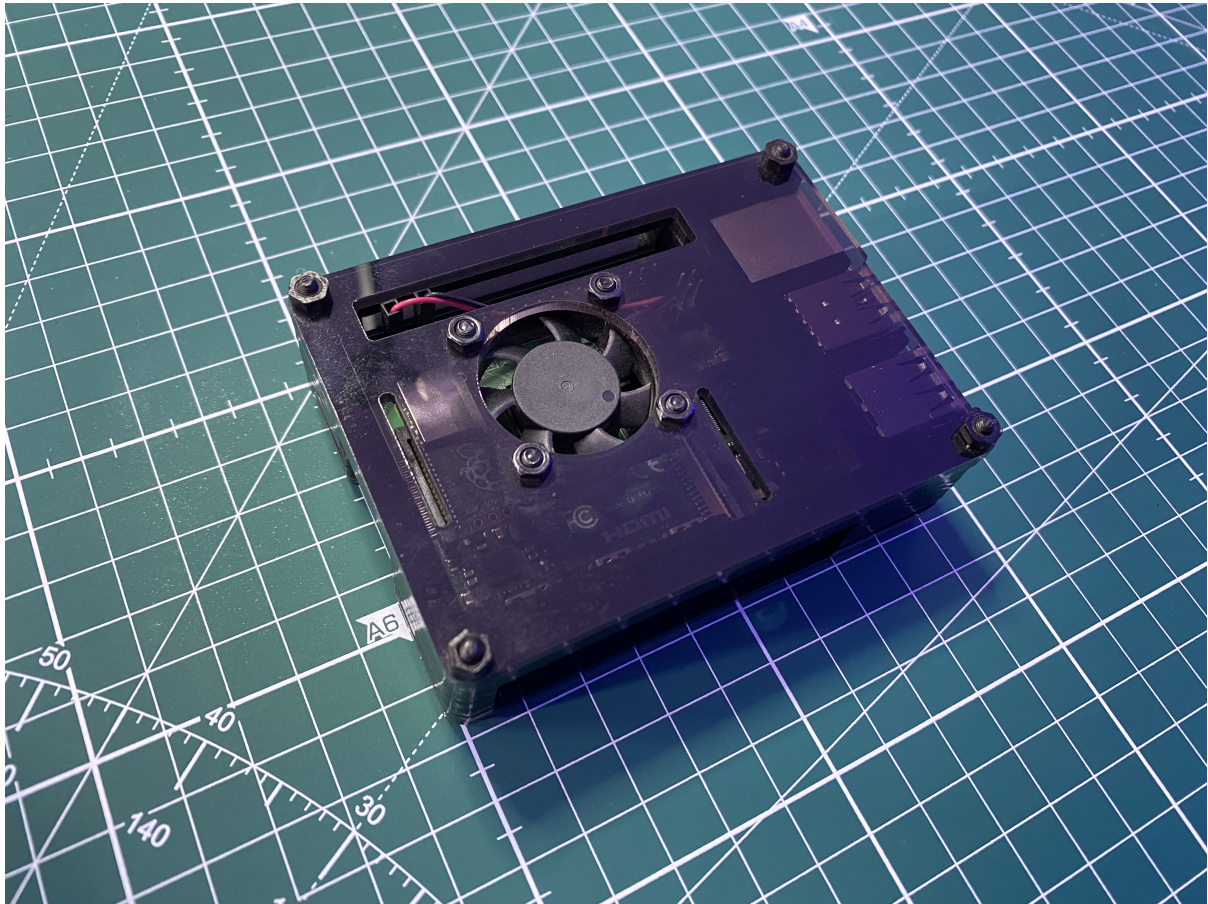
- [PlatformIO](#) IDE, which is powerful and cool at the same time: it allows to perform static code analysis and neatly manages projects and libraries
- [UIFlow](#), a graphical environment (web-based and/or local app) that's programmable in Blockly and [MicroPython](#)

Needless to say, I'll go for the first choice. I clearly remember the time when IDEs didn't exist (yes, I'm *that* old) and all you got from a compile-link-run session was a disturbing message that read "segmentation fault (core dump)". We now have modern and productive environments, and - overall - choice, so pick up your environment of choice and follow the rest of this articles as a reference.



In addition to the ESP32 family, we'll use an ESP8266-based **smart badge** that will act as a wearable device.

And, of course, we can't help but use the ubiquitous [Raspberry Pi](#) - that year over year is getting specs almost on-par with his bigger cousins - to act as physical and logical link between edge and the Cloud environments. This pocketable Linux device will be crucial in bridging the local MQTT instance to the OCI Cloud instance described and set up in the *previous chapter*.



The Raspberry Side: MQTT Bridging

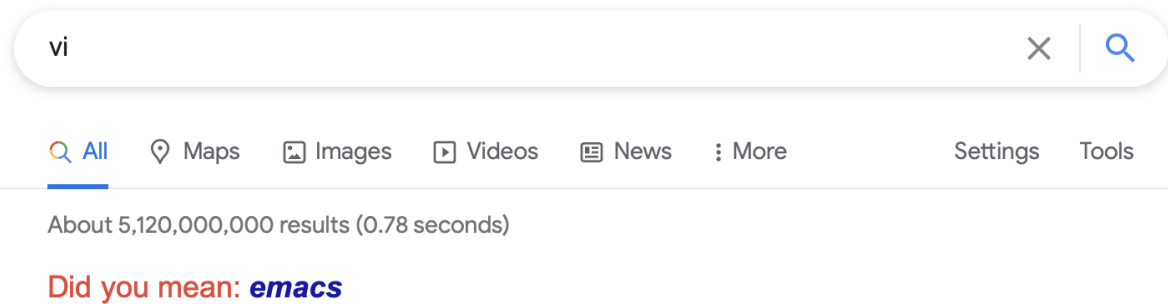
Installing **Mosquitto** and the related CLI utilities on a Pi is straightforward, by issuing the command `sudo apt install mosquitto` and `sudo apt install mosquitto-clients`. Once started, you can check the status by issuing the command `systemctl status mosquitto`, which should be followed by something like:

```
Loaded: loaded (/lib/systemd/system/mosquitto.service; enabled; vendor
preset: enabled)
Active: active (running) since Tue 2021-03-30 17:22:35 CEST; 19h ago
   Docs: man:mosquitto.conf(5)
        man:mosquitto(8)
Main PID: 635 (mosquitto)
   Tasks: 1 (limit: 4915)
  CGroup: /system.slice/mosquitto.service
         └─635 /usr/sbin/mosquitto -c /etc/mosquitto/mosquitto.conf

...

```

and proceed to modify the `/etc/mosquitto/conf.d/mosquitto.conf` file configuring the bridging mechanism. Most of the default parameters are just fine (unless you want to setup an encrypted connection between microcontrollers and the edge instance). In our case we'll just configure the bridge, so using our favorite editor of choice, even if your favorite search engine suggests otherwise(!):



and reaching the Bridges section:

```
# =====
# Bridges
# =====

# A bridge is a way of connecting multiple MQTT brokers together.
# Create a new bridge using the "connection" option as described below.
Set
# options for the bridges using the remaining parameters. You must specify
the
# address and at least one topic to subscribe to.
```

we can add the following parameters:

```
connection proxima
address <host:port>
topic # out 0 "" edge/
topic alarm in 0 cloud/ edge/
```

where the **host** and **port** parameters are the public IP address and the of the OCI instance we configured in the first episode, and the other parameters indicate that:

- in the first line, we're going to relay **all** messages to the Cloud instance (prefixed by the **edge/** parameter), so all the messages issued on the the edge in the topic **device/machine/x** will be processed by the Cloud Mosquitto as **edge/device/machine/x**.
- in the second line, we'll receive any message from the Cloud in the topic **alarm**, specifying the **cloud/** prefix and locally processed in topic **cloud/alarm**.

Sure enough, we also need to setup the certificate based SSL/TLS support, so reach for the section regarding security and complete it with:

```
# -----
# Certificate based SSL/TLS support
# -----
```

```

# Either bridge_cafile or bridge_capath must be defined to enable TLS
support
# for this bridge.
# bridge_cafile defines the path to a file containing the
# Certificate Authority certificates that have signed the remote broker
# certificate.
# bridge_capath defines a directory that will be searched for files
containing
# the CA certificates. For bridge_capath to work correctly, the
certificate
# files must have ".crt" as the file ending and you must run "openssl
rehash
# <path to capath>" each time you add/remove a certificate.
#bridge_capath
bridge_cafile /etc/mosquitto/certs/ca.crt

# Path to the PEM encoded client certificate, if required by the remote
broker.
bridge_certfile /etc/mosquitto/certs/server.crt

# Path to the PEM encoded client private key, if required by the remote
broker.
bridge_keyfile /etc/mosquitto/certs/server.key

# When using certificate based encryption, bridge_insecure disables
# verification of the server hostname in the server certificate. This can
be
# useful when testing initial server configurations, but makes it possible
for
# a malicious third party to impersonate your server through DNS spoofing,
for
# example. Use this option in testing only. If you need to resort to using
this
# option in a production environment, your setup is at fault and there is
no
# point using encryption.
bridge_insecure true

```

thus creating a `certs` directory under `/etc/mosquitto` and copying the `ca.cert`, `server.crt` and `server.key` files we generated during the first episode in section **Secure the MQTT Server running on OCI Compute**.

That is easy to test. Issuing a listening command to the Cloud instance in a shell, as shown in the first episode:

```

mosquitto_sub -d -t '#' -h <your host> -u <username> -P <password> -p
<port> --insecure --cafile certs/ca.crt --cert certs/server.crt --key
certs/server.key

```

and sending a message to the local Raspberry Pi

```
mosquitto_pub -h <your RPi IP address> -t test -m 'Sympathetic resonance'
```

we should receive on the Cloud Mosquitto shell the message:

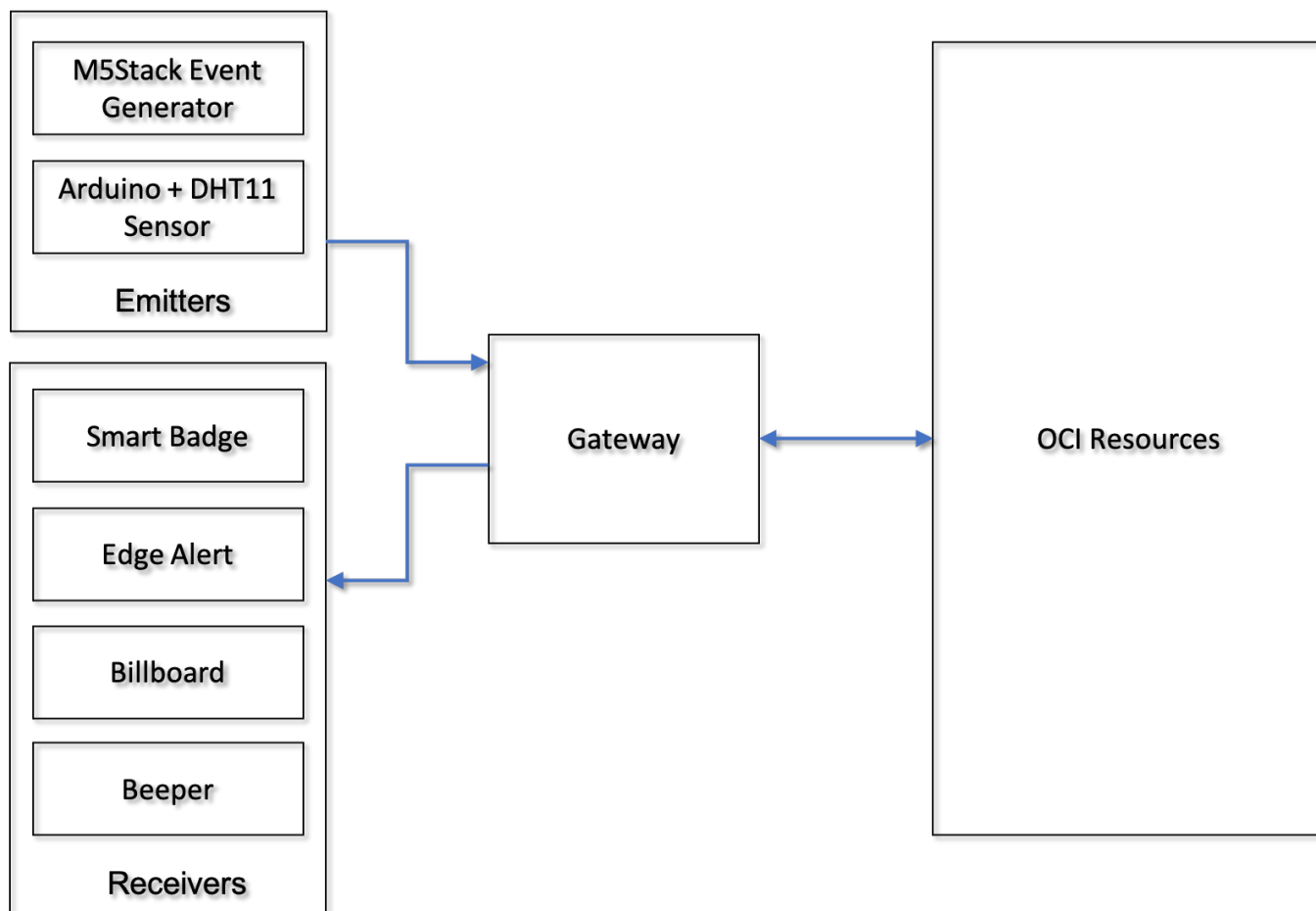
```
Client (null) received PUBLISH (d0, q0, r0, m0, 'edge/testtopic', ... (21 bytes))
Sympathetic resonance
```

showing that the two thingies are effectively talking themselves - albeit in a single direction, for now.

The pipelines we'll design in Stream Analytics will provide the logic to test the bidirectional dialogue. And, now, let's have some healthy fun with sensors!

Edge Programming

The goal is to build an edge that can easily fit into a small briefcase, and - certainly - an ESP32-based kit will help saving space, time, and power consumption. Let's consider a setup that includes some edge emitters (MQTT publishers), some receivers (MQTT subscribers) and the Gateway:



Publishers

- A multipurpose ***ESP32** development board that can be used to generate single or burst messages to trigger actions within Stream Analytics. Connecting a detection sensors - of course - would be a lot better, but I'd find hard to simulate a gathering alert within my workshop room. Firing the messages directly from the board has a clear advantage over messages sent from a shell or a software simulator: we could connected a sensor to the board and maintain the same codebase.
- An **Arduino MKR1000** connected to a DHT11 sensor. I tried to use an Env Hat based on the BMP80 sensor attached to a M5Stick-C but that proved to be extremely unreliable in terms of data measurement: the temperature and humidity detected are heavily affected by the heat generated from the M5Stick-C, so I'll perservere with an old-fashioned configuration encompassing the Arduino and a simpler, cheaper DHT11.

You can find all the sources I've used at this link (NOTE: insert the GitHub link, open in a new window/tab).

Subscribers

- An **ESP8266-based Smart Badge** that will change a text status message whenever something relevant has been detected and analysed in OCI.
- An **ESP32 Edge Alert device** that could be eventually connected to some actuators to show the alerts and make people aware about the occuring anomaly.
- An **ESP32 Billboard** based on **M5Paper**, a small but powerful e-ink tablet that will show the sequence of alerts coming from OCI.
- An **ESP32 Wearable** and/or pocketable Beeper.

Both the publisher and the subscriber will use the **PubSubClient API**. Specifically, the Publishers will send messages to the local MQTT server via the **publish** method:

```
Result = mqttClient.publish(MACHINE_TOPIC, msg, true);
M5.Lcd.setCursor(10, 60);
if (Result)
    M5.Lcd.println("Sent.");
else
    M5.Lcd.println("Not sent.");
```

while Subscribers will initialize the callback in the **setup()** portion of the code (executed only once at startup):

```
configTime(gmtOffset_sec, daylightOffset_sec, ntpServer);
timestamp = getTime();
if (timestamp > 0)
    noTime = false;
```

```
mqttClient.subscribe(TOPIC);
Serial.println("Subscribed!");

mqttClient.setCallback(DisplayCallback);
delay(100);
```

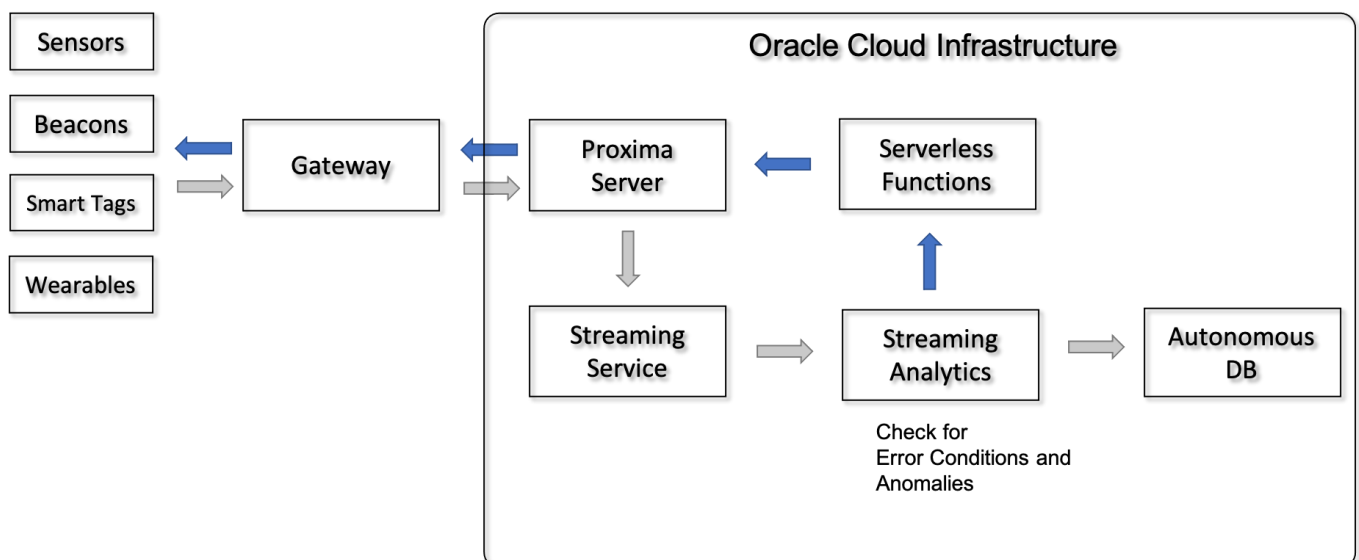
and upon the reception of new messages (in our case, from OCI), processing will occur in

```
void DisplayCallback(char* topic, byte* payload, unsigned int len)
{
    // Process message
    // Serial.println((String)topic);
}
```

Programming these gizmos is fun and it's a very effective means of spreading the culture of programming among students of all levels (including myself). Plus, there's plenty of examples available on the Web.

Still, we need to design a way to return alarm messages from Stream Analytics to the edge, using the MQTT Bridge feature we set up not too long ago.

As described in the previous Episode, our approach will be as the following:



thus we (thankfully) need to tinker with [Oracle Functions](#).

Serverless Time!

FnProject is a cool Open Source *serverless platform* that can scale from microdevices to megainstallations, launched in **2017**, and later transformed and evolved in an industrial-strength OCI service called **Oracle Functions**.

Developing a function in OCI requires either:

- Preparing your environment and use the handy OCI Cloud Shell, as shown [here](#)
- or using your [local machine](#) as a development environment, that involves:
 - a **Signing Key**
 - a **Profile**
 - a valid **Docker** installation
 - the Fn Project **CLI**
 - an **OCI Context**
 - setting the Context **setup**
 - an **Auth Key**
 - using **docker login** to store the function in Oracle Cloud Infrastructure Registry as a **docker image**

Either way, you'll be good to go with the function deployment in OCI.

We will use a Custom Dockerfile to build our image in Python, such as the following:

```
FROM fnproject/python:3.6-dev as build-stage

WORKDIR /function

ADD requirements.txt /function/

RUN pip3 install --target /python/ --no-cache --no-cache-dir -r
requirements.txt && rm -fr ~/.cache/pip /tmp* requirements.txt func.yaml
Dockerfile .venv

ADD . /function/
RUN rm -fr /function/.pip_cache

FROM fnproject/python:3.6

WORKDIR /function

COPY --from=build-stage /python /python
COPY --from=build-stage /function /function
COPY certs /function

ENV PYTHONPATH=/function:/python

ENTRYPOINT ["/python/bin/fdk", "/function/func.py", "handler"]
```

specifying Python requirements in **requirements.txt** file as we're going to use the Paho Library:

```
fdk
paho-mqtt
```

and write some code to complete the round trip, copying the `certs` folder and files used to access the MQTT Server on OCI in the function directory. Please find the Dockerfile and the code at this address (NOTE: insert the GitHub link, open in a new window/tab).

Oracle Functions (as **Fn Project**) requires the function to be installed in an artifact called **Application**, a logical grouping of functions, which can be created via the `fn` CLI (specifying the OCI subnets) or in the OCI Web console following the path **Home » Developer Services » Functions**:

The screenshot shows the Oracle Cloud console interface for creating a new application. The left sidebar shows the navigation menu with 'Functions' selected. The main content area is titled 'New Application'. The form includes the following fields and options:

- Name:** GabbaMQTT
- VCN in train19:** Oracle GoldenGate Stream Analytics VCN (with a link to 'Change Compartment')
- Subnets in train19:** Oracle GoldenGate Stream Analytics Subnet (Regional) (with a link to 'Change Compartment')
- Tagging:** A section explaining that tagging is a metadata system for organizing and tracking resources. It includes a table with columns for Tag Namespace, Tag Key, and Value. The Tag Namespace is currently set to 'None (add a free-form tag)'. There is a '+ Additional Tag' button.

At the bottom of the form, there are 'Create' and 'Cancel' buttons.

Once the application is created, we can deploy the function (this time we'll leverage *the good-ole CLI*) using

```
fn build
fn deploy --app <app name>
```

where you can see some familiar Docker (layer-related) output messages and the result of deployment.

```
Building image fra.ocir.io/emeaseitalyproxima/gabba-
repository/mqtt_pub:0.0.2 .
Parts: [fra.ocir.io emeaseitalyproxima gabba-repository mqtt_pub:0.0.2]
Pushing fra.ocir.io/emeaseitalyproxima/gabba-repository/mqtt_pub:0.0.2 to
docker registry...The push refers to repository
[fra.ocir.io/emeaseitalyproxima/gabba-repository/mqtt_pub]
77ff3ee9cb37: Pushed
```

```

43353efa4559: Pushed
0f6cdd7e71a8: Layer already exists
3697bae2d860: Layer already exists
0b66d6c41076: Layer already exists
85e1ba76ed69: Layer already exists
6881daa7bad0: Layer already exists
7352730c981f: Layer already exists
9d95bea46bad: Layer already exists
b84a8d46e8fb: Layer already exists
f66ed577df6e: Layer already exists
0.0.2: digest:
sha256:e82a0abc009c0a132fc6c3c35fc8d88f516589b35a96907c41e41a350619872d
size: 2626
Updating function mqtt_pub using image
fra.ocir.io/emeaseitalyproxima/gabba-repository/mqtt_pub:0.0.2...

```

The status of the function will be reflected in the OCI Web Console as well as in CLI, issuing the command `fn list functions <app name>`:







```

NAME                IMAGE                                                    ID
mqtt_pub            fra.ocir.io/emeaseitalyproxima/gabba-repository/mqtt_pub:0.0.2
ocid1.fnfunc.oc1.eu-frankfurt-
1.aaaaaaaaabbknysfrffi2olayuzkykcv5boop72qi75k5aqqjwjfdlycutq

```

The function must be provided with the three input parameters that can be set on the OCI Web Console in the **Configuration** submenu:

Configuration

Key	Value	
<input type="text"/>	<input type="text"/>	+
mqtt_username	Your Username	 
mqtt_password	Your Password	 
mqtt_topic	edge/alarm	 

Showing 3 items

specifying your Mosquitto username, password and the alarm topic `edge/alarm`. Note those parameters, as we'll use them to perform some smoke test!

Creating the API Gateway and an API deployment

The mechanisms to expose and consume APIs in Oracle Cloud Infrastructure are accessible in the **Main menu » Developer Services » API Management** section of OCI Web Console. We'll create an API Gateway first, and then an API deployment specifying the Oracle Function we created previously. Creating an API Gateway involves specifying:

- the **Name** (duh!)
- the **Type** (public/private)
- the **Virtual Cloud Network** (the one created by Stream Analytics, or your own)
- the **Subnet** (ditto)

Gateways

Create Gateway

NAME
ProximaSafe

TYPE
Private

COMPARTMENT
train19
ociobtmeaa (root)/train19

CUSTOM DNS ⓘ

VIRTUAL CLOUD NETWORK IN TRAIN19 (CHANGE COMPARTMENT)
Oracle GoldenGate Stream Analytics VCN

SUBNET IN TRAIN19 (CHANGE COMPARTMENT)
Oracle GoldenGate Stream Analytics Subnet

TAGS
Tagging is a metadata system that allows you to organize and track resources within your tenancy. Tags are composed of keys and values that can be attached to resources.
[Learn more about tagging](#)

TAG NAMESPACE	TAG KEY	VALUE
None (add a free-form tag)		

+ Additional Tag

Create Cancel

Hitting the blue **Create** button starts the magic, and creation is quick. Then, we can proceed to shape our API deployment by clicking the link named "Deployments" in the bottom left *Resources* section and fire the **Create Deployment** procedure, which consists of three stages:

- Providing the name, the prefix, and additional Policies such as Authentication, CORS, Rate Limiting (quite important if API is exposed to the cruel external world)

From Scratch

Deploy a custom API using the provided form ✓

Upload an existing Deployment API

Upload a JSON file containing the API

Basic Information

NAME

PATH PREFIX ⓘ

COMPARTMENT

ociobtemea (root)/train19

API Request Policies

Authentication

Configures authentication Add

- Provide the Routing logic with the path, method (we'll use POST), the type (Oracle Functions), the Application and the Function name we previously deployed

Route 1

PATH ⓘ

METHODS

TYPE

Specifies the type of the backend service. [Learn more](#) about the Oracle Functions backend.

APPLICATION IN TRAIN19 [\(CHANGE COMPARTMENT\)](#)

FUNCTION NAME

> [Show Route Request Policies](#)

> [Show Route Response Policies](#)

> [Show Route Logging Policies](#)

- Carefully review your API configuration and submit!

This configuration will be managed by Stream Analytics as a **Target**, used at the end of an analysis pipeline to invoke the function and send an alarm to the edge.

The combination of *Sources* (messages from the edge routed to OCI Streaming), *Patterns* already available for phenomena detection and *Targets* (Autonomous DB, Serverless Functions, Kafka Endpoints) will be the triad (in musical context) to be tinkered designing new use cases and discover valuable information not previously available.

Next Episode - Use cases and Stream Analytics pipelines

Going further, we'll need to design some simple use cases as an example and develop some pipelines within Stream Analytics to close the loop and test our setup. See you on the next chapter.

Zip and Zest!