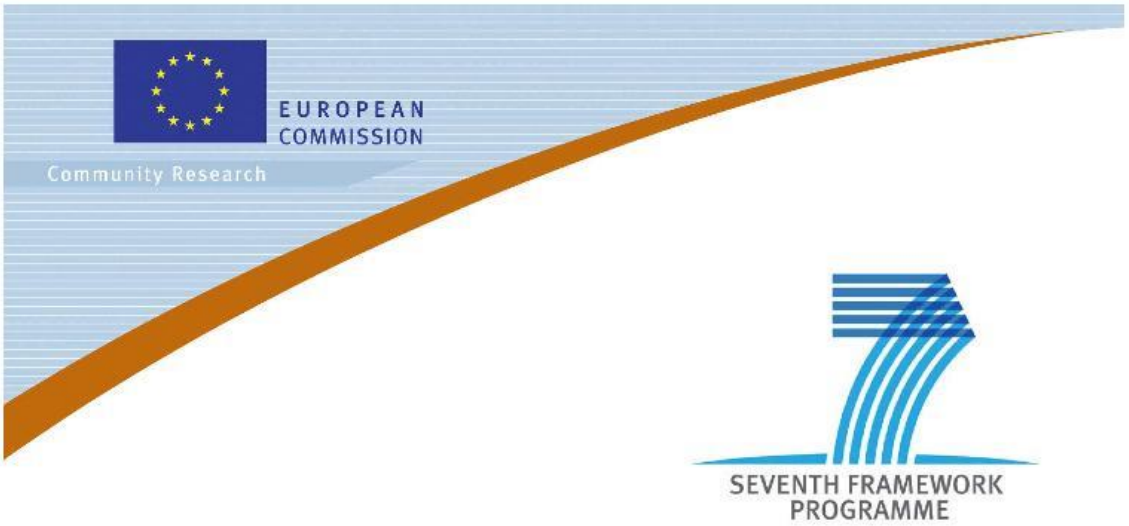

Private Public Partnership Project (PPP)

Large-scale Integrated Project (IP)

D.11.1.3: FIWARE Tour Guide

Project acronym: FI-Core
Project full title: Future Internet - Core
Contract No.: 632893
Strategic Objective: FI.ICT-2011.1.7 Technology foundation: Future Internet Core Platform
Project Document Number: ICT-2013-FI-632893-WP11-D.11.1.3
Project Document Date: 2016-10-25
Deliverable Type and Security: PU
Author: José Manuel Cantera (TID)
Contributors: Alberto Martín (Bitergia), Pablo Fernández (ULPGC)

Executive Summary

This report describes the activities performed in order to maintain and evolve the FIWARE Tour Guide for developers. The FIWARE Tour Guide is a tutorial document and accompanying application which allows developers to get started with FIWARE in a short timeframe.

During the period covered by the present document the activities which have been conducted are the following:

- Refine the contents of the Tour Guide by adding new sections or updating existing ones.
- Develop a new version of the “Tour Guide Application” making it easier for new learners. The outcome is a new release available (Release 1.0).
- Align extra Tour Guide examples with the content and structure of the Tour Guide Application.
- Coordinate all the teams involved in contributing to the Tour Guide, including chapter leaders and GE development teams.
- Disseminate the Tour Guide among the FIWARE Developers Community by using it for training purposes.
- Define a roadmap for the Tour Guide so that it is always kept alive and aligned with the latest FIWARE platform advances.

Table of Contents

1 Introduction

1.1 Introduction

This document describes all the activities targeted to maintain and evolve the FIWARE Tour Guide for developers. The FIWARE Tour Guide is continuously evolving, thus this document summarizes both the activities which have been completed at the time of writing, together with a development roadmap to be conducted by the FIWARE Community.

1.2 Target audience

The target audience of the present document is anyone who wants to know more about the *FIWARE Tour Guide for developers*. In addition, any member of the FIWARE Community will benefit from reading it, as it allows to understand the efforts made by FIWARE to lower the entry barriers.

1.3 Related readings

The document on Friendliness Activities (D11.9.1 Report on FIWARE Friendliness activities) is closely related to this document, as it describes other complementary activities aimed at improving FIWARE adoption by developers. There is a certain degree of overlap between both and the present document. However, the latter provides a closer and more detailed view on a concrete aspect exclusively: the FIWARE Tour Guide.

2 Tour Guide Application

2.1 Introduction

When the first version of the FIWARE Tour Guide was created, it was based on some fictitious use cases that facilitated the description of the main functionalities offered by the different GEs or Chapters in FIWARE. Soon it was detected the necessity of enabling direct experimentation with the technologies to improve learning processes. As a result, it was decided to build a tutorial application aimed at supporting the Tour Guide use cases, enabling developers to follow a "try and tweak" learning process.

The *FIWARE Tour Guide Application* is a reference application aimed at teaching and demonstrating how to combine different Generic Enablers (GEs), in order to create a smart context-aware application. It exploits the capabilities offered by Docker Containers provided by FIWARE GEris. Furthermore, the application allows an incremental instantiation and linkage of different FIWARE GEs, as it is based on docker compose.

2.2 Approach

The first step taken to build the FIWARE Tour Guide application was to think up a rich use case: smart management of a big restaurant franchise, including reservations, reviews and all the parameters which have to do with managing each restaurant on a day by day basis. Afterwards a complete specification of requirements was drafted (included in this document as an annex) to be used as a reference for the development team.

Once the requirements were ready, a decision on the supporting technologies was made. At this point it was clear that Docker technology was perfect for this purpose, as it simplifies the deployment and configuration processes. Furthermore, it enables a modular composition (docker compose), which is crucial for separating the different chapters and GEs in FIWARE. This choice was reinforced by the fact that the FIWARE Technical Committee agreed on making the provision of docker containers mandatory for each GEri.

Another important point was to define a bootstrap process that would enable to load initial data into the application. For this purpose, data coming from the Open Euskadi (*Basque Country Government*) open data portal was used. Data from restaurants located at the Basque Country Region in Spain were loaded. All this bootstrapping process was automated for the benefit of the end user who will be able to try and tweak with mock data.

Finally, it was decided to give priority to the integration of those GEs which have to do with Internet of Things (IoT) scenarios, i.e. Orion, IDAS and CKAN, together with those GEs which implement transversal security layers.

The application has been developed following the FIWARE Developer Guidelines, including peer reviews and controlled landing of new features. The development of the application has been integrated into the FIWARE sprint plannings and Jira. As any other FIWARE software artefact, it

includes unit tests and automated build procedures so that to ensure that no new landing code breaks existing functionalities.

2.3 Results

A detailed tutorial on using and learning the Tour Guide Application is continously updated at http://fiwaretourguide.readthedocs.io/en/latest/fiware-tour-guide-application-a-tutorial-on-how-to-integrate-the-main-fiware-ges/introduction/.

Additionally, the following subsections describe in more detail the results achieved, and provides a good technical overview of the work undertaken.

2.3.1 Application functionalities

The list below summarizes the main functionalities currently offered by the smart restaurant (Tour Guide) application:

- Different user profiles (customer, restaurant manager, franchise manager) and functionalities per profile
- Admit Customer reservations in accordance with current occupation and reservations made.
- Register customer reviews according to different criteria (service, food, etc.).
- Real-time control of different parameters at each restaurant location (occupation, temperature, etc.).
- Short time historic data of the different parameters monitored. Publication of open data concerning the most relevant information about the different restaurant locations.
- Web user interface to monitor information about restaurants.

2.3.2 Architecture

The figure below describes the architecture of the Smart Restaurant application, which integrates a number of GEs:

- **IoT GEs**: *Backend Device Management - IDAS*. It is in charge of connecting IoT devices (temperature & humidity) using the UL20 client. This component translates UL20 client requests into NGSI context entities, enabling querying and subscribing to sensor data.
- **Data GEs**: *Orion Context Broker*. It is responsible for managing all the application context information modelled as NGSI entities (Restaurant, Reservation, Review, …). *Cygnus*, part of the Cosmos ecosystem, is responsible for persisting historical context data in a target backend (MySQL or Hadoop) or as open data (*CKAN*). *Cygnus* is connected to *Orion Context Broker* through the subscription/notification interface.
- **Security GEs**:
    - *Authorization PDP - AuthZForce,* provides an API to get authorization decisions based on authorization policies.

- *IDM KeyRock* covers a number of aspects involving user profile management, OAuth authentication, authorization & trust management, Single Sign-On (SSO) to service domains and identify federation towards applications. It interacts with *AuthZForce*.

Finally, through a front-end application, managers get access to restaurants under his duty and restaurant customers can make and browse reviews, or even ask for reservations.

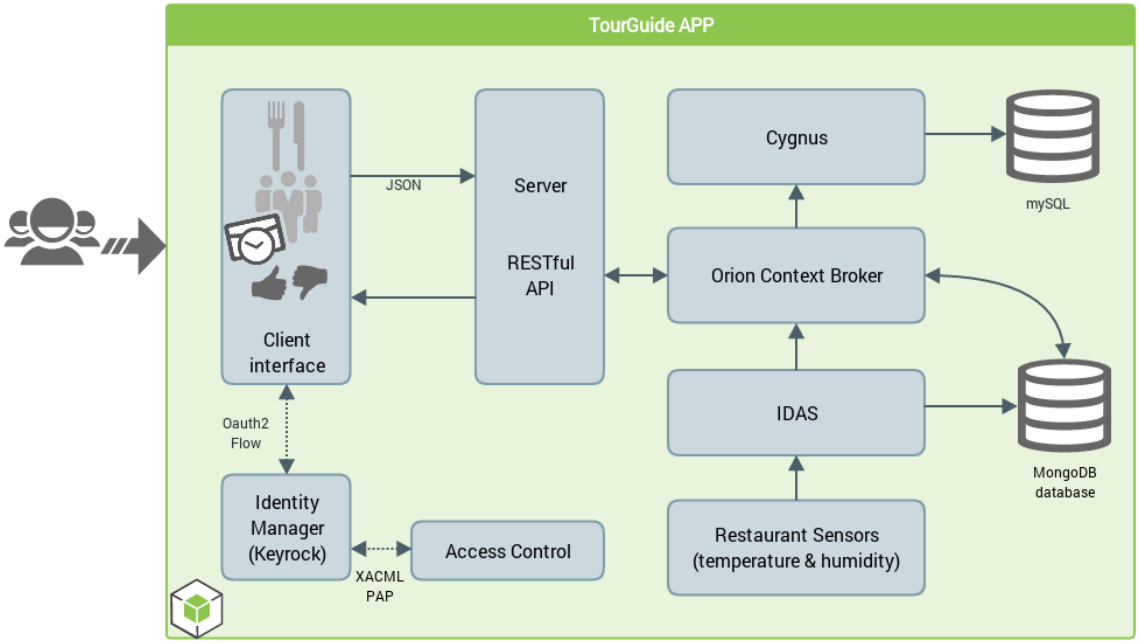

Figure 1.- Architecture of the Tour Guide Application (release 1.0)

2.3.3 Implementation approach

To this aim we have connected, using Docker-compose, different GEs deployed on different containers. Docker-compose enables the creation of a complete environment just by writing a .yml (or .yaml) schema file. In that file several parameters are defined: containers to be created, volumes to share data, how to link containers, ports exposed and environment variables that will be used to configure the Generic Enablers. The docker-compose file created for the Tour Guide Application defines the scenario described by the image below. This configuration has all the advantages of using isolated environments and exposing each generic enabler from a desired port.

The Tour Guide Application uses the official Docker from the GEris and has contributed to debugging or polishing them.

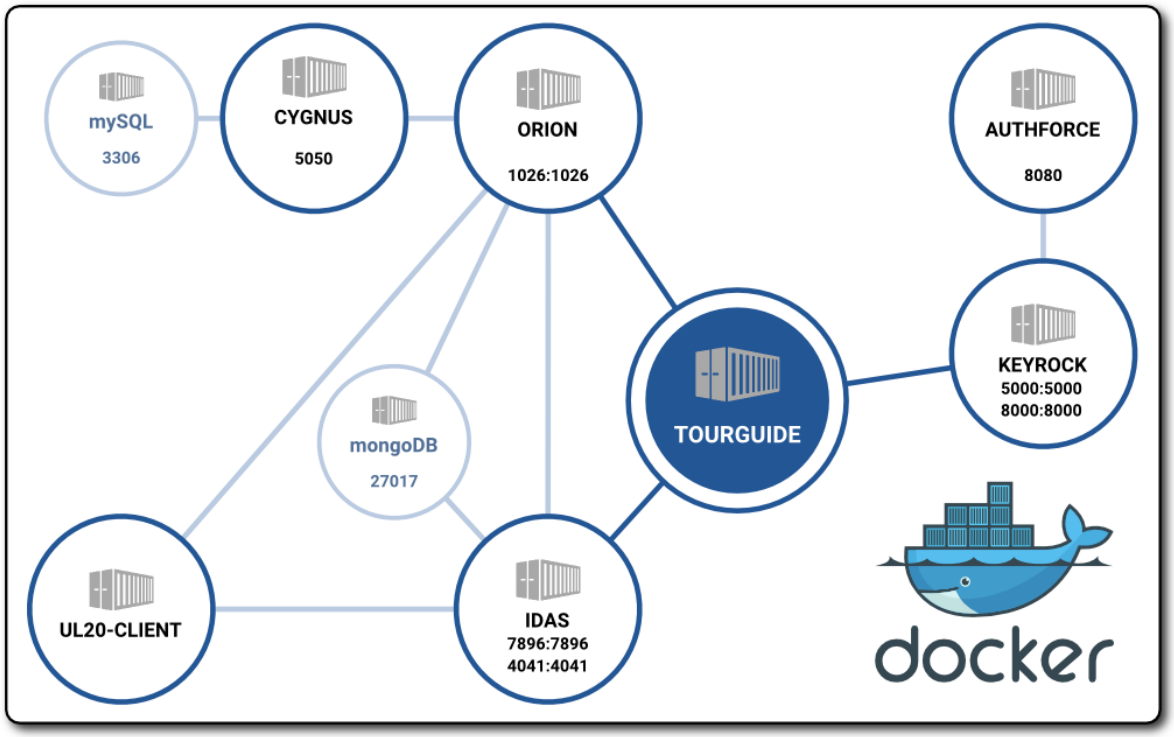

Figure 2.- Docker container structure (reléase 1.0)

2.3.4 Development

As it was reported in previous issues of this document the application code has been developed using Node.js and HTML5 technologies. The Travis CI tool is used for running all tests and ensuring quality. The Github repository used is http://github.com/fiware/tutorials.TourGuide-App. The development process follows usual Github workflows with pull requests, reviews and code landing.

Additionally, the development has been integrated into the usual agile process followed by FIWARE, performing sprint plannings and management of tasks through Jira. It is noteworthy that FIWARE offers support to this application through the ask.fiware.org platform.

Last but not least it is important to remark that FIWARE has been able to create a real community of developers around this Tour Guide Application. Developers from Bitergia, TID and the University of Las Palmas (ULPGC) are contributing to the code on a daily basis. It is our aspiration that contributions coming from other members of the FIWARE Community happen, particularly in the context of the upcoming FIWARE Bounty Programme.

3 FIWARE Tour Guide Application Step by Step

3.1 Introduction

This chapter describes, step by step, how to use and combine the different components of the FIWARE stack integrated by the Tour Guide Application.

The content of this chapter has been exported to the readthedocs site of the FIWARE Tour Guide. It is intended to be the main reference of tutorial content in FIWARE.

3.2 Managing Context Information

In this section the steps needed to run Orion Context Broker and some example operations are detailed. Further information about Orion Context Broker can be found in the official documentation at http://fiware-orion.readthedocs.io/en/latest/.

For running an Orion instance ready to be used, run:

```
docker-compose up orion
```

or using the Command Line Interface (CLI):

```
./tour-guide start orion
```

Note that in order to use the sample data, the `Fiware-Service: tourguide` HTTP header has to be added to every request.

In order to perform a request to the Context Broker we need to know its hostname. You can modify the system hosts file using the CLI provided in the Tour Guide Application by executing `./tour-guide configure hosts`. After executing the configure hosts command, you can perform requests to the context broker using `orion` instead of its IP.

You can check that Orion is running and the hosts file is correctly configured by issuing the following HTTP request:

```
GET orion:1026/version
```

Then, you will obtain a JSON response similar to:

```
{
    "orion": {
        "version": "1.3.0",
        "uptime": "0 d, 0 h, 2 m, 42 s",
        "git_hash": "cb6813f044607bc01895296223a27e4466ab0913",
        "compile_time": "Fri Sep 2 08:36:02 UTC 2016",
        "compiled_by": "root",
        "compiled_in": "ba19f7d3be65"
    }
}
```

If you want to create a new restaurant, you can issue a request with restaurant information in the payload body and specifying a franchise using the `Fiware-ServicePath` HTTP header.

For instance, in order to create a restaurant belonging to `Franchise1` the following HTTP request can be used:

```
POST orion:1026/v2/entities/

Headers: {
    'Content-Type': 'application/json',
    'Fiware-Service': 'tourguide'
    'Fiware-ServicePath': '/Franchise1'
}

{
    "id": "sample-id",
    "type": "Restaurant",
    "address": {
        "type": "PostalAddress",
        "value": {
            "streetAddress": "Cuesta de las Cabras Aldapa 2",
            "addressRegion": "Araba",
            "addressLocality": "Alegría-Dulantzi",
            "postalCode": "01240"
        }
    },
    "aggregateRating": {
        "type": "AggregateRating",
        "value": {
            "ratingValue": 3,
            "reviewCount": 98
        }
    },
    "capacity": {
        "type": "PropertyValue",
        "value": 100
    },
    "department": {
        "type": "Text",
        "value": "Franchise1"
    },
    "description": {
        "type": "Text",
        "value": "Sample description"
    },
    "location": {
        "type": "geo:point",
        "value": "42.8404625, -2.5123277"
    },
    "name": {
        "type": "Text",
        "value": "Sample-restaurant"
    },
    "occupancyLevels": {
        "type": "PropertyValue",
        "value": 0,
        "metadata": {
            "timestamp": {
                "type": "DateTime",
                "value": "2016-09-19T06:32:15.901Z"
            }
        }
    },
    "priceRange": {
```

```
        "type": "Number",
        "value": 0
    },
    "telephone": {
        "type": "Text",
        "value": "945 400 868"
    }
}
```

Afterwards, you can retrieve the restaurant data just created using its id. The `keyValues` option is used in order to get a more compact and brief representation, including just attribute names and values:

```
GET orion:1026/v2/entities/sample-id?options=keyValues

Headers : {
    'Content-Type': 'application/json',
    'Fiware-Service': 'tourguide',
    'Fiware-ServicePath': '/Franchise1'
}
```

You should get:

```
{
    "id": "sample-id",
    "type": "Restaurant",
    "address": {
        "streetAddress": "Cuesta de las Cabras Aldapa 2",
        "addressRegion": "Araba",
        "addressLocality": "Alegría-Dulantzi",
        "postalCode": "01240"
    },
    "aggregateRating": {
        "ratingValue": 3,
        "reviewCount": 98
    },
    "capacity": 100,
    "department": "Franchise1",
    "description": "Sample description",
    "location": "42.8404625, -2.5123277",
    "name": "Sample-restaurant",
    "occupancyLevels": 0,
    "priceRange": 0,
    "telephone": "945 400 868"
}
```

If you need to update restaurant data after refurbishing it, which may imply a change in capacity and description values, you can use the following HTTP request:

```
PATCH orion:1026/v2/entities/sample-id/attrs?options=keyValues

Headers {
    'Content-Type': 'application/json',
    'Fiware-Service': 'tourguide'
    'Fiware-ServicePath': '/Franchise1'
}


{
```

```
    "description": "New sample description",
    "capacity": 150
}
```

You can check that the values have been updated by retrieving the attributes with:

```
GET orion:1026/v2/entities/sample-id/attrs?attrs=description,capacity&options=keyValues

Headers {
    'Content-Type': 'application/json',
    'Fiware-Service': 'tourguide',
    'Fiware-ServicePath': '/Franchise1'
}
```

It should return:

```
{
    "description": "New sample description",
    "capacity": 150
}
```

3.3 Managing IoT Data

With the Tour Guide application, you can generate, simulate and give persistence to data coming from IoT devices. For that purpose, we generate humidity and temperature (virtual) sensors for the kitchen and dining room of restaurants. All the data provided by sensors will be propagated to Cygnus, using MySQL database by default.

Note that, to this aim, we will need to previously load the Restaurants information. For further information on how to generate them just execute the load command with the `--help` parameter:

```
$ ./tour-guide load --help
```

And make sure the required services (`tourguide, orion, idas and cygnus`) are up and running. By executing:

```
$ ./tour-guide start tourguide
```

It will start all the services needed that were still not running.

Once done, we can start working with sensors.

3.3.1 Working with sensors

In Tour Guide Application we provide different ways to work with sensors:

```
$ ./tour-guide sensors
Usage: tour-guide sensors [-h | --help] <command> <options>

Run sensors related commands:

  create                    Create sensors for the restaurants available in the
application.
  update                    Update all restaurant sensors measurements.
  send-data                 Send a single measurement for a specific sensor.
  simulate-data             Simulate a sensor sending data over a period of time.

Command options:

  -h  --help                Show this help.

Use 'tour-guide sensors <command> --help' to get help about a specific <command>.
```

3.3.2 Creating sensors

Creating sensors is as simple as running:

```
$ ./tour-guide sensors create
```

This will create and initialize four sensors for each of the available restaurants in the application. There will be temperature and relative humidity sensors for both the kitchen and the dining room of the restaurant. This command executes a script (sensorsgenerator.js) that uses the IoT agent API to create the sensors and initialize them with a default measurement.

Similarly, if you want to create the sensors yourself, you can follow these steps:

Register a new service with the IoT Agent

The first step is to register a new service configuration with the IoT Agent (if it doesn't exists). To do this we need to send a POST request to `http://localhost:4041/iot/services` with the following JSON payload:

```
{
    "services": [{
        "apikey": "tourguide-devices",
        "cbroker": "http://orion:1026",
        "resource": "/iot/d",
        "entity_type": "Restaurant"
    }]
}
```

We must include the following HTTP headers when sending the request:

- Fiware-Service, that for TourGuide-App will have a value of `tourguide`,
- Fiware-ServicePath, that will be the organization of the restaurant we want to register sensors for, e.g. `/Franchise1`, as this will allow us to register sensors for restaurants that belong in that organization,
- Content-type of the data we are POSTing, this will be `application/json`.

Now we can do the request:

```
$(curl -H 'content-type: application/json' -H 'fiware-service: tourguide' -H 'fiware-
servicepath: /Franchise1' -X POST 'http://localhost:4041/iot/services' -d @- ) << EOF
{
    "services": [{
        "apikey": "tourguide-devices",
        "cbroker": "http://orion:1026",
        "resource": "/iot/d",
        "entity_type": "Restaurant"
    }]
}
EOF
```

The response should be `{}` if there was no error and the log of the IoT Agent should have entries like this:

```
time = 2016 - 09 - 29 T11: 12: 35.920 Z | ... | srv = tourguide | subsrv = /Franchise1 |
msg=Request for path [/iot/services] from [localhost:4041] | comp=IoTAgent
time = 2016 - 09 - 29 T11: 12: 35.920 Z | ... | srv = tourguide | subsrv = /Franchise1 |
msg=Body: {
    "services": [{
        "apikey": "tourguide-devices",
        "cbroker": "http://orion:1026",
        "resource": "/iot/d",
        "entity_type": "Restaurant"
    }]
} | comp = IoTAgent time = 2016 - 09 - 29 T11: 12: 35.923 Z | ... | srv = tourguide | sub
srv = /Franchise1 | msg=Creating new set of 1 services | comp=IoTAgent
time = 2016 - 09 - 29 T11: 12: 35.923 Z | ... | srv = tourguide | subsrv = /Franchise1 |
msg=Looking for entity params ["resource","apikey"] | comp=IoTAgent
time = 2016 - 09 - 29 T11: 12: 35.930 Z | ... | srv = tourguide | subsrv = /Franchise1 |
msg=Device group for fields [["resource","apikey"]] not found: [{"resource":"/iot/d","api
key":"tourguide-devices"}] | comp=IoTAgent
```

```
time = 2016 - 09 - 29 T11: 12: 35.935 Z | ... | srv = tourguide | subsrv = /Franchise1 |
msg=Storing device group with id [57ecf7235755b6010061870a], type [Restaurant], apikey [t
ourguide-devices] and resource [/iot/d] | comp=IoTAgent
```

This information will be stored in a MongoDB database. We can check this with:

```
$ docker exec -i -t mongodb mongo
MongoDB shell version: 2.6.11
connecting to: test
Welcome to the MongoDB shell.
For interactive help, type "help".
For more comprehensive documentation, see
    http://docs.mongodb.org/
Questions? Try the support group
    http://groups.google.com/group/mongodb-user
```

We can list the available databases and select the one used by the IoT Agent, `iotagentul`.

```
> show dbs;
admin       (empty)
iotagentul  0.031GB
local       0.031GB
orion       0.031GB
> use iotagentul;
switched to db iotagentul
```

In this database we have two collections, `devices` and `groups`. The service we have just registered should be in the `groups` collection.

```
> show collections;
devices
groups
system.indexes
> db.groups.find();
{ "_id" : ObjectId("57ecf7235755b6010061870a"), "subservice" : "/Franchise1", "service" :
"tourguide", "type" : "Restaurant", "apikey" : "tourguide-devices", "resource" :
"/iot/d", "staticAttributes" : [ ], "__v" : 0 }
```

As we can see, the service has been stored. Next is to register the sensors.

3.3.3 Registering a new sensor

To register a new temperature sensor, we need to send a POST request to `http://localhost:4041/iot/devices` with the following JSON payload:

```
{
    "devices": [{
        "device_id": "0115206c51f60b48b77e4c937835795c33bb953f-kitchen-temperature",
        "entity_name": "0115206c51f60b48b77e4c937835795c33bb953f",
        "protocol": "UL20",
        "entity_type": "Restaurant",
        "timezone": "Europe/Madrid",
        "attributes": [{
            "object_id": "t",
            "name": "temperature:kitchen",
            "type": "Number"
        }]
    }]
}
```

`entity_name` will be the Id of the restaurant to which we want to add the sensor, and `device_id` will be the Id of the sensor. In this example, we'll use the restaurant with Id `0115206c51f60b48b77e4c937835795c33bb953f`. The Id of the sensor will be a compound of the restaurant Id, the room and the type of the sensor, like `0115206c51f60b48b77e4c937835795c33bb953f-kitchen-temperature`.

As with the service, we need to add the following HTTP headers in our request:

- Fiware-Service, that for TourGuide-App will have a value of `tourguide`,
- Fiware-ServicePath, that will be the organization of the restaurant, in our example `/Franchise1`,
- Content-type of the data we are POSTing, this will be `application/json`.

We do the request:

```
$(curl -v -H 'content-type: application/json' -H 'fiware-service: tourguide' -H 'fiware-
servicepath: /Franchise1' -X POST 'http://localhost:4041/iot/devices' -d @- ) << EOF
{
    "devices": [{
        "device_id": "0115206c51f60b48b77e4c937835795c33bb953f-kitchen-temperature",
        "entity_name": "0115206c51f60b48b77e4c937835795c33bb953f",
        "protocol": "UL20",
        "entity_type": "Restaurant",
        "timezone": "Europe/Madrid",
        "attributes": [{
            "object_id": "t",
            "name": "temperature:kitchen",
            "type": "Number"
        }]
    }]
}
EOF
```

and the response should be `{}` if there was no error. The log of the IoT Agent should have entries like this:

```
time = 2016 - 09 - 29 T12: 55: 54.992 Z | ... | srv = tourguide | subsrv = /Franchise1 |
msg=Request for path [/iot/devices] from [localhost:4041] | comp=IoTAgent
time = 2016 - 09 - 29 T12: 55: 54.992 Z | ... | srv = tourguide | subsrv = /Franchise1 |
msg=Body: {
    "devices": [{
        "device_id": "0115206c51f60b48b77e4c937835795c33bb953f-kitchen-temperature",
        "entity_name": "0115206c51f60b48b77e4c937835795c33bb953f",
        "protocol": "UL20",
        "entity_type": "Restaurant",
        "timezone": "Europe/Madrid",
        "attributes": [{
            "object_id": "t",
            "name": "temperature:kitchen",
            "type": "Number"
        }]
    }]
} | comp = IoTAgent
time = 2016 - 09 - 29 T12: 55: 54.993 Z | ... | srv = tourguide | subsrv = /Franchise1 |
msg=Handling device provisioning request. | comp=IoTAgent
time = 2016 - 09 - 29 T12: 55: 54.995 Z | ... | srv = tourguide | subsrv = /Franchise1 |
msg=Looking for entity with id [0115206c51f60b48b77e4c937835795c33bb953f-kitchen-
temperature]. | comp=IoTAgent
```

```
time = 2016 - 09 - 29 T12: 55: 54.997 Z | ... | srv = tourguide | subsrv = /Franchise1 |
msg=Entity [0115206c51f60b48b77e4c937835795c33bb953f-kitchen-
temperature] not found. | comp=IoTAgent
time = 2016 - 09 - 29 T12: 55: 54.997 Z | ... | srv = tourguide | subsrv = /Franchise1 |
msg=Looking for entity params ["service","subservice","type"] | comp=IoTAgent
time = 2016 - 09 - 29 T12: 55: 54.999 Z | ... | srv = tourguide | subsrv = /Franchise1 |
msg=Registering device into NGSI Service: {
    "id": "0115206c51f60b48b77e4c937835795c33bb953f-kitchen-temperature",
    "type": "Restaurant",
    "name": "0115206c51f60b48b77e4c937835795c33bb953f",
    "service": "tourguide",
    "subservice": "/Franchise1",
    "active": [{
        "object_id": "t",
        "name": "temperature:kitchen",
        "type": "Number"
    }],
    "staticAttributes": [],
    "lazy": [],
    "commands": [],
    "timezone": "Europe/Madrid",
    "protocol": "UL20",
    "internalId": null,
    "subscriptions": []
} | comp = IoTAgent time = 2016 - 09 - 29 T12: 55: 55.000 Z | ... | srv = tourguide | sub
srv = /Franchise1 | msg=No Context Provider registrations found for unregister | comp=IoT
Agent
time = 2016 - 09 - 29 T12: 55: 55.109 Z | ... | srv = tourguide | subsrv = /Franchise1 |
msg=Initial entity created successfully. | comp=IoTAgent
time = 2016 - 09 - 29 T12: 55: 55.110 Z | ... | srv = tourguide | subsrv = /Franchise1 |
msg=Storing device with id [0115206c51f60b48b77e4c937835795c33bb953f-kitchen-
temperature] and type [Restaurant] | comp=IoTAgent
time = 2016 - 09 - 29 T12: 55: 55.120 Z | ... | srv = tourguide | subsrv = /Franchise1 |
msg=Device provisioning request succeeded | comp=IoTAgent
```

We can check the mongo database again, this time using the `devices` collection:

```
$ docker exec -i -t mongodb mongo
MongoDB shell version: 2.6.11
connecting to: test
> use iotagentul;
switched to db iotagentul
> db.devices.find();
{ "_id" : ObjectId("57ed0f5b5755b6010061870b"), "protocol" : "UL20", "internalId" : null,
"subservice" : "/Franchise1", "service" : "tourguide", "name" :
"0115206c51f60b48b77e4c937835795c33bb953f", "type" : "Restaurant", "id" :
"0115206c51f60b48b77e4c937835795c33bb953f-kitchen-temperature", "creationDate" :
ISODate("2016-09-29T12:55:55.109Z"), "subscriptions" : [ ], "active" : [ { "type" :
"Number", "name" : "temperature:kitchen", "object_id" : "t" } ], "__v" : 0 }
```

As we can see, the device has been registered with the IoT Agent. There is one more check we can do, as we are adding the sensor to an existing entity in Orion. We can do a request to orion for the restaurant and see if the new attribute has been added:

```
$ curl -s -H 'Fiware-Service: tourguide'
http://localhost:1026/v2/entities/0115206c51f60b48b77e4c937835795c33bb953f | json_reforma
t
    {
        "id": "0115206c51f60b48b77e4c937835795c33bb953f",
        "type": "Restaurant",
        "address": {
            "type": "PostalAddress",
```

```
            "value": {
                "streetAddress": "Cuesta de las Cabras Aldapa 2",
                "addressRegion": "Araba",
                "addressLocality": "Alegría-Dulantzi",
                "postalCode": "01240"
            },
            "metadata": {}
        },
        ...
        "temperature:kitchen": {
            "type": "Number",
            "value": " ",
            "metadata": {}
        }
    }
```

As we can see, a new attribute `temperature:kitchen` has been added to the restaurant entity. Please note that the value is empty, as we have not sent any measurement yet.

3.3.4 Providing new sensor data

Once we have our sensors registered, we can begin sending measurements. We can do this with the `tourguide` CLI by running:

```
$ ./tour-guide  sensors  update
```

This will get the current value of the sensors, do a small variation and send a new measurement for each of the available sensors. If we want to do this ourselves, we can send new measurements using the HTTP Ultralight 2.0 protocol. See Send data below for a detailed description on how to do this.

If we want to update a single sensor, we can use the `send-data` command of `tourguide` CLI to do that:

```
$ ./tour-guide sensors send-data --help
Usage: tour-guide sensors send-data [-h | --help] [-i <sensorId> | --sensor-id
<sensorId>]
                                    [ -d <ul20-string> | --data <ul20-string> ]

Send a single measurement for a specific sensor using a Ultralight 2.0 string.

Command options:

  -h  --help                Show this help.

Required parameters:

  -i  --sensor-id  <sensorId>   The sensor Id to modify.  The Id format is
'<restaurantId>-<room>-<type>', with
                              <restaurantId> being the Id of the restaurant where the
sensor is located,
                              <room> the room of the restaurant: kitchen, diner,
                              <type> the type of the sensor: temperature,
relativeHumidity.
  -d  --data <ul20-string>      The string to send with the new measurement.  Examples of
this are:
                              't|20' for temperature (20 C),
                              'h|0.4' for relativeHumidity (40%).
```

Here, we must specify the sensor Id and the new measurement using a Ultralight 2.0 string. Continuing with our example, imagine we want to send a temperature of 25 degrees for the sensor we created before. The sensor Id was `0115206c51f60b48b77e4c937835795c33bb953f-kitchen-temperature`, and the data string we must send is `t|25`:

```
$ ./tour-guide sensors send-data -i 0115206c51f60b48b77e4c937835795c33bb953f-kitchen-
temperature -d 't|25'
```

If the request is successful, there will be no output. If instead of using the `tourguide` CLI we want to send the measurement ourselves, we can do so by sending a POST request to `http://localhost:7896/iot/d` and add the following parameters:

- k=${api_key}
- i=${sensor_id}

Here, the `${api_key}` value is the one we defined when registering the service. In our example it is `tourguide-devices`. The `${sensor_id}` is the Id of the sensor we want to update. In our example it is `0115206c51f60b48b77e4c937835795c33bb953f-kitchen-temperature`.

We also need to add the `Content-Type: text/plain` HTTP header to our request and send the Ultralight 2.0 data string as our payload:

```
curl -v -X POST -H 'content-type: text/plain'
'http://localhost:7896/iot/d?k=tourguide-
devices&i=0115206c51f60b48b77e4c937835795c33bb953f-kitchen-temperature' -d 't|25'
```

There will be no output if the request is successful. We can then check the attribute to see if the value has been updated:

```
$ curl -s -H 'Fiware-Service: tourguide'
http://localhost:1026/v2/entities/0115206c51f60b48b77e4c937835795c33bb953f/attrs/temperat
ure:kitchen  | json_reformat
    {
        "type": "Number",
        "value": "25",
        "metadata": {}
    }
```

As we can see, the value of the `temperature:kitchen` attribute has been updated to `25`.

3.3.5 Simulating multiple sensor data

Finally, we can simulate a sensor behavior, by sending measurements of a sensor periodically by using the `simulate-data` command of `tourguide` CLI:

```
$ ./tour-guide sensors simulate-data --help
Usage: tour-guide sensors simulate-data [-h | --help] [-i <sensorId> | --sensor-id
<sensorId>]
                                        [ -t <type> | --type <type> ] [ -d <n> | --delay
<n> ]

Simulate a sensor periodically sending data.
```

```
Command options:

  -h  --help                Show this help.

Required parameters:

  -i  --sensor-id  <sensorId>   The sensor Id to modify.  The Id format is
'<restaurantId>-<room>-<type>', with
                              <restaurantId> being the Id of the restaurant where the
sensor is located,
                              <room> the room of the restaurant: kitchen, diner,
                              <type> the type of the sensor: temperature,
relativeHumidity.
  -t  --type <type>           Type of the sensor.  This must be the same type specified
in the sensor Id.
                              Valid values: temperature, relativeHumidity.
  -d  --delay <n>             Delay in seconds between sensor readings.
```

This command will get the current value of the specified sensor, add or subtract a small value (between 0 and 5) and update the sensor, repeating the process at the specified intervals. Following our example, we can do:

```
$ ./tour-guide sensors simulate-data -i 0115206c51f60b48b77e4c937835795c33bb953f-kitchen-
temperature -t ‘temperature’ -d 30
```

This will send a new `temperature` measurement of the `0115206c51f60b48b77e4c937835795c33bb953f-kitchen-temperature` sensor every `30` seconds. To stop sending measurements, interrupt the command with `Control + C`. This command uses the same method described in the previous section for sending new measurements. We can check the current value stored on the restaurant entity as before or use the following request to get just the value of the attribute:

```
curl -s -H 'Fiware-Service: tourguide'
http://localhost:1026/v2/entities/0115206c51f60b48b77e4c937835795c33bb953f/attrs/temperat
ure:kitchen | json_reformat
    {
        "type": "Number",
        "value": "25",
        "metadata": {}
    }
```

3.4 Configuring Security Aspects

3.4.1 Keyrock (Identity Manager)

Keyrock is a Generic Enabler integrated in the Tour Guide Application, aware of the user profile management, authorization and authentication among others.

For testing purposes, we have generated a set of users, organizations, apps, roles and permissions to be loaded automatically in Keyrock. To load them, we just need to run the following:

```
$ ./tour-guide configure keyrock
```

This will load all the information in Keyrock, and automatically sync with Authzforce, the Generic Enabler aware of storing the XACML policies.

Once the information is loaded, we will need to get the Oauth credentials from Keyrock and add them to the Tour Guide Application configuration by doing:

```
$ ./tour-guide configure oauth
```

This step can be done also manually. You can go to the Keyrock interface:

```
http://keyrock:8000
```

And authenticate with a user with the application `provider` role (in this application example, the user `pepproxy@test.com` listed below). There, select the application `TourGuide` already registered, and you will find there the Oauth credentials (`client ID` and `client SECRET`).

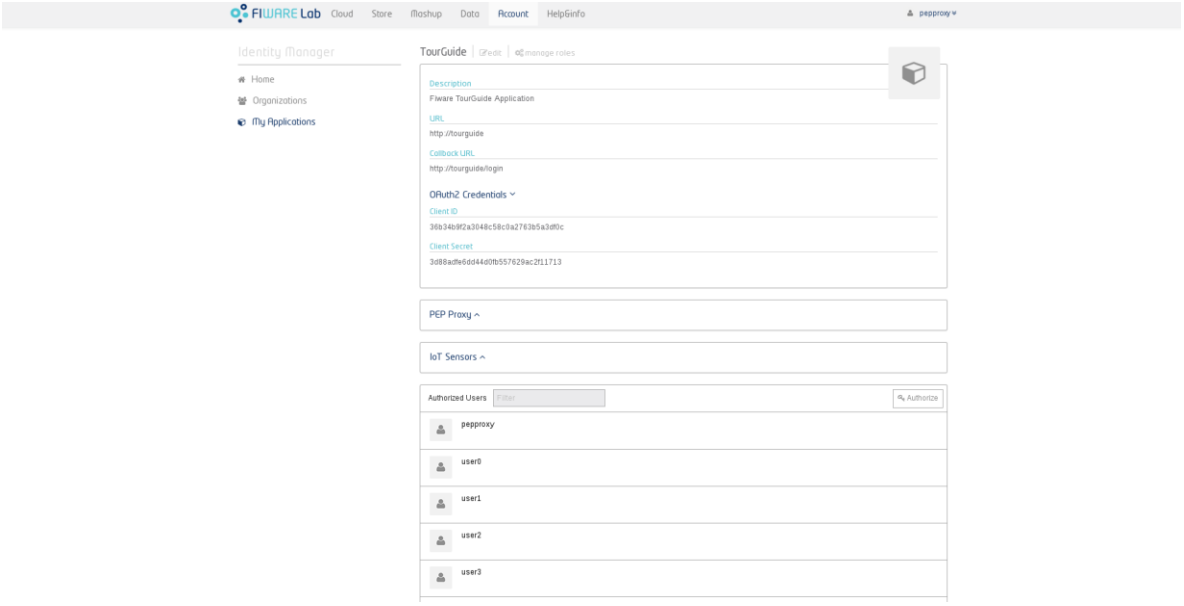

Figure 2b .- Keyrock application settings

Once you get them, you will need to modify the `config.js` file inside the `tourguide` container.

Probably the easiest way is accessing the container:

```
docker exec -it tourguide /bin/bash
```

And there, modify the parameters `config.clientId` and `config.clientSecret`:

```
vi tutorials.TourGuide-App/server/config.js
```

Finally, still inside the container, we should reload apache:

```
service apache2 reload
```

Users roles and permissions

The file with all the information related to the set of users, organizations, apps, roles and permissions is available here.

Note: the following provision is intended just for testing purposes. Check the full Keyrock API description here.

Users

The default set of users provided is described below.

Role	Username	Password
Admin	idm	idm
Provider	pepproxy@test.com	test
Owner	user0@test.com	test
Owner	user1@test.com	test
Owner	user2@test.com	test
Owner	user3@test.com	test
Owner	user4@test.com	test
Owner	user5@test.com	test
Owner	user6@test.com	test
Owner	user7@test.com	test
Owner	user8@test.com	test
Owner	user9@test.com	test

Once generated, you can retrieve the whole list by using Keyrock SCIM 2.0 REST API:

```
curl -X GET -H "Content-Type: application/json" -H "X-auth-token: ADMIN"
"http://keyrock:5000/v3/OS-SCIM/v2/Users/"
```

And you will see an output like:

```
{
    "totalResults": 13,
    "Resources": [{
        "userName": "idm",
        "urn:scim:schemas:extension:keystone:2.0": {
            "domain_id": "default"
        },
        "active": true,
```

```
        "id": "idm_user"
    }, {
        "userName": "user0@test.com",
        "urn:scim:schemas:extension:keystone:2.0": {
            "domain_id": "default"
        },
        "active": true,
        "id": "user0"
    }, {
        "userName": "user1@test.com",
        "urn:scim:schemas:extension:keystone:2.0": {
            "domain_id": "default"
        },
        "active": true,
        "id": "user1"
    }, {
        "userName": "user2@test.com",
        "urn:scim:schemas:extension:keystone:2.0": {
            "domain_id": "default"
        },
        "active": true,
        "id": "user2"
    }...],
    "schemas": ["urn:scim:schemas:core:2.0", "urn:scim:schemas:extension:keystone:2.0"]
}
```


Or generate users yourself, as explained here.

Applications

Organizations are also known as *projects* if using the Identity API.

Besides the Organizations that Keyrock automatically creates, four Organizations as Franchises have been provided.

Organization name	Description	Users
Franchise1	Franchise1	user0@test.com (owner)
Franchise2	Franchise2	user0@test.com (owner)
Franchise3	Franchise3	user0@test.com (owner)
Franchise4	Franchise4	user0@test.com (owner)

You can list all of the organizations using:

```
curl -X GET -H "Content-Type: application/json" -H "X-auth-token: ADMIN"
"http://keyrock:5000/v3/OS-SCIM/v2/Organizations/"
```

This will display the organizations generated:


```
{
    "totalResults": 27,
    "Resources": […{
        "active": true,
        "urn:scim:schemas:extension:keystone:2.0": {
            "domain_id": "default"
        },
        "description": "Test Franchise1",
        "name": "Franchise1",
```

```
        "id": "f3aa9a45d1174b32a178dd281e801fd8"
    }, ... {
        "active": true,
        "urn:scim:schemas:extension:keystone:2.0": {
            "domain_id": "default"
        },
        "description": "Test Franchise4",
        "name": "Franchise4",
        "id": "06a127d2a7534500bb5fb17b5d54d308"
    }],
    "schemas": ["urn:scim:schemas:core:2.0", "urn:scim:schemas:extension:keystone:2.0"]
}
```

Find here how to generate organizations.

Applications

We have registered a Consumer (or App) in Keyrock.

Application name	Description	URL	Redirect URI
FIWARE TourGuide	Fiware TourGuide Test Application	http://tourguide	http://tourguide/login

You can list them all by running:

```
curl -X GET -H "Content-Type: application/json" -H "X-auth-token: ADMIN"
"http://keyrock:5000/v3/OS-OAUTH2/consumers/"
```

And the output:

```
{
    "links": {
        "self": "http://keyrock:5000/v3/OS-OAUTH2/consumers",
        "previous": null,
        "next": null
    },
    "consumers": [{
        "scopes": [],
        "redirect_uris": [],
        "description": "Application that acts as the IdM itself. To see the administratio
n section of the web portal grant provider to a user in this application.",
        "links": {
            "self": "http://keyrock:5000/v3/OS-OAUTH2/consumers/idm_admin_app"
        },
        "extra": {
            "is_default": true
        },
        "is_default": true,
        "client_type": "confidential",
        "response_type": "code",
        "grant_type": "authorization_code",
        "id": "idm_admin_app",
        "name": "idm_admin_app"
    }, {
        "scopes": ["all_info"],
        "pep_proxy_name": "pep_proxy_7479c6d8886a4b1db211bd76fda1c1f6",
        "redirect_uris": ["http://tourguide/login"],
        "name": "TourGuide",
        "img": "/static/dashboard/img/logos/small/app.png",
        "extra": {
```

```
            "url": "http://tourguide",
            "pep_proxy_name": "pep_proxy_7479c6d8886a4b1db211bd76fda1c1f6",
            "iot_sensors": [],
            "ac_domain": "zgUcVoWDEea5lAJCrBEABw",
            "img": "/static/dashboard/img/logos/small/app.png"
        },
        "url": "http://tourguide",
        "ac_domain": "zgUcVoWDEea5lAJCrBEABw",
        "links": {
            "self": "http://keyrock:5000/v3/OS-
OAUTH2/consumers/36b34b9f2a3048c58c0a2763b5a3df0c"
        },
        "iot_sensors": [],
        "response_type": "code",
        "client_type": "confidential",
        "grant_type": "authorization_code",
        "id": "36b34b9f2a3048c58c0a2763b5a3df0c",
        "description": "Fiware TourGuide Application"
    }]
}
```


Or generate your own as explained here.

Roles

The following list shows the roles generated:

Role name	Granted to user
Provider	pepproxy@test.com
End user	All
Franchise Manager	user0@test.com (Franchise1)
Franchise Manager	user1@test.com (Franchise2)
Franchise Manager	user2@test.com (Franchise3)
Franchise Manager	user3@test.com (Franchise4)
Global Manager	user0@test.com

You can retrieve them by executing the following query:

```
curl -X GET -H "Content-Type: application/json" -H "X-auth-token: ADMIN"
"http://keyrock:5000/v3/OS-ROLES/roles/"
```

Generating the following output:


```
{
    "links": {
        "self": "http://keyrock:5000/v3/OS-ROLES/roles",
        "previous": null,
        "next": null
    },
    "roles": [{
        "is_internal": true,
        "application_id": "idm_admin_app",
        "id": "provider",
        "links": {
            "self": "http://keyrock:5000/v3/OS-ROLES/roles/provider"
        },
        "name": "Provider"
```

```
    }, {
        "is_internal": true,
        "application_id": "idm_admin_app",
        "id": "purchaser",
        "links": {
            "self": "http://keyrock:5000/v3/OS-ROLES/roles/purchaser"
        },
        "name": "Purchaser"
    }, {
        "is_internal": false,
        "application_id": "36b34b9f2a3048c58c0a2763b5a3df0c",
        "id": "17d245ab695847f1800df8f85b360df9",
        "links": {
            "self": "http://keyrock:5000/v3/OS-
ROLES/roles/17d245ab695847f1800df8f85b360df9"
        },
        "name": "End user"
    }, {
        "is_internal": false,
        "application_id": "36b34b9f2a3048c58c0a2763b5a3df0c",
        "id": "a5b6a9daa0594f8d818e3a83da5a498e",
        "links": {
            "self": "http://keyrock:5000/v3/OS-
ROLES/roles/a5b6a9daa0594f8d818e3a83da5a498e"
        },
        "name": "Franchise manager"
    }, {
        "is_internal": false,
        "application_id": "36b34b9f2a3048c58c0a2763b5a3df0c",
        "id": "0efd09a12f074f63abe53ee943cfa6f5",
        "links": {
            "self": "http://keyrock:5000/v3/OS-
ROLES/roles/0efd09a12f074f63abe53ee943cfa6f5"
        },
        "name": "Global manager"
    }]
}
```


Or generate some as explained here.

Permissions

Permissions can be listed by doing:

```
curl -X GET -H "Content-Type: application/json" -H "X-auth-token: ADMIN"
"http://keyrock:5000/v3/OS-ROLES/permissions/"
```

Getting the following output:


```
{
    "links": {
        "self": "http://keyrock:5000/v3/OS-ROLES/permissions",
        "previous": null,
        "next": null
    },
    "permissions": [{
        "xml": null,
        "resource": null,
        "name": "Manage the application",
        "links": {
            "self": "http://keyrock:5000/v3/OS-ROLES/permissions/manage-application"
        },
        "is_internal": true,
        "action": null,
```

```
        "application_id": "idm_admin_app",
        "id": "manage-application"
    }, {
        "xml": null,
        "resource": null,
        "name": "Manage roles",
        "links": {
            "self": "http://keyrock:5000/v3/OS-ROLES/permissions/manage-roles"
        },
        "is_internal": true,
        "action": null,
        "application_id": "idm_admin_app",
        "id": "manage-roles"
    }, {
        "xml": null,
        "resource": null,
        "name": "Get and assign all public application roles",
        "links": {
            "self": "http://keyrock:5000/v3/OS-ROLES/permissions/get-assign-public-
roles"
        },
        "is_internal": true,
        "action": null,
        "application_id": "idm_admin_app",
        "id": "get-assign-public-roles"
    }, {
        "xml": null,
        "resource": null,
        "name": "Manage Authorizations",
        "links": {
            "self": "http://keyrock:5000/v3/OS-ROLES/permissions/manage-authorizations"
        },
        "is_internal": true,
        "action": null,
        "application_id": "idm_admin_app",
        "id": "manage-authorizations"
    }, {
        "xml": null,
        "resource": null,
        "name": "Get and assign only public owned roles",
        "links": {
            "self": "http://keyrock:5000/v3/OS-ROLES/permissions/get-assign-public-owned-
roles"
        },
        "is_internal": true,
        "action": null,
        "application_id": "idm_admin_app",
        "id": "get-assign-public-owned-roles"
    }, {
        "xml": null,
        "resource": null,
        "name": "Get and assign all internal application roles",
        "links": {
            "self": "http://keyrock:5000/v3/OS-ROLES/permissions/get-assign-internal-
roles"
        },
        "is_internal": true,
        "action": null,
        "application_id": "idm_admin_app",
        "id": "get-assign-internal-roles"
    }, {
        "xml": null,
        "resource": "NGSI10/queryContext?limit=1000&entity_type=reservation",
        "name": "reservations",
        "links": {
```

```
            "self": "http://keyrock:5000/v3/OS-
ROLES/permissions/1c9af9da448a41f1ae5682930d2f59c0"
        },
        "is_internal": false,
        "action": "POST",
        "application_id": "36b34b9f2a3048c58c0a2763b5a3df0c",
        "id": "1c9af9da448a41f1ae5682930d2f59c0"
    }, {
        "xml": null,
        "resource": "NGSI10/queryContext?limit=1000&entity_type=review",
        "name": "reviews",
        "links": {
            "self": "http://keyrock:5000/v3/OS-
ROLES/permissions/7a26fd22c9ba4495802c7cf6683e4cdd"
        },
        "is_internal": false,
        "action": "POST",
        "application_id": "36b34b9f2a3048c58c0a2763b5a3df0c",
        "id": "7a26fd22c9ba4495802c7cf6683e4cdd"
    }, {
        "xml": null,
        "resource": "NGSI10/queryContext?limit=1000&entity_type=restaurant",
        "name": "restaurants",
        "links": {
            "self": "http://keyrock:5000/v3/OS-
ROLES/permissions/ea915f7a7e654536aa3c587f58ce83df"
        },
        "is_internal": false,
        "action": "POST",
        "application_id": "36b34b9f2a3048c58c0a2763b5a3df0c",
        "id": "ea915f7a7e654536aa3c587f58ce83df"
    }]
}
```


Or you can generate them yourself as explained here.

3.4.2 AuthZForce (Policy Decision Point)

Authzforce policies are generated automatically by Keyrock based on the default provision file.

By running:

```
$ ./tour-guide configure keyrock
```

Policies are generated and synchronized with Authzforce. To be able to query the Authzforce container, we will need to add the container IP to our hostfile. This can be achieved by doing (sudo required):

```
sudo ./tour-guide configure hosts -m
```

After that you will be able to query the Authzforce container to check the policies generated.

First getting the domain where the policies are stored:

```
curl -s --request GET \
http://authzforce:8080/authzforcece/domains | awk '/href/{print $NF}' | cut -d '"' -f2
```

Will give us something like:

```
_lczKYmCEeaNFgJCrBEACA
```

Secondly, retrieving the list of policies id's stored:

```
curl -s --request GET \
http://authzforce:8080/authzforce-ce/domains/{$DOMAIN}/pap/policies |
 xmllint --format -
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<resources xmlns="http://authzforce.github.io/rest-api-model/xmlns/authz/5"
xmlns:ns2="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
xmlns:ns3="http://authzforce.github.io/core/xmlns/pdp/3.6"
xmlns:ns4="http://www.w3.org/2005/Atom" xmlns:ns5="http://authzforce.github.io/pap-dao-
flat-file/xmlns/properties/3.6">
  <ns4:link rel="item" href="b0654ddd-e74a-4f4f-8f91-d81470af70a1"/>
  <ns4:link rel="item" href="root"/>
</resources>
```

And selecting one of the policies, we can get the versions stored of this policy:

```
curl -s --request GET \
http://authzforce:8080/authzforce-ce/domains/{$DOMAIN}/pap/policies/{$POLICY_ID}" |
 xmllint --format -
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<resources xmlns="http://authzforce.github.io/rest-api-model/xmlns/authz/5"
xmlns:ns2="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
xmlns:ns3="http://authzforce.github.io/core/xmlns/pdp/3.6"
xmlns:ns4="http://www.w3.org/2005/Atom" xmlns:ns5="http://authzforce.github.io/pap-dao-
flat-file/xmlns/properties/3.6">
  <ns4:link rel="item" href="1.0"/>
</resources>
```

Finally, choosing one of the versions, we can get the full policy set:

```
curl -s --request GET \
http://authzforce:8080/authzforce-
ce/domains/{$DOMAIN}/pap/policies/{$POLICY_ID}/{$VERSION} | xmllint --format -
```

3.5 Publishing historical data

This section assumes that you already have some sensors working with with the Tour Guide Application. If you have not do so, please see Managing IoT data above.

In order to persist data from Orion, we will be using the Cygnus connector to send and store the data on a MySQL database.

3.5.1 Cygnus configuration

The default configuration uses a MySQL container with the default database and `root` user with `mysql` as password. If you want to change this configuration, you can use the `tourguide configure cygnus` command to change these credentials.

```
$ ./tour-guide configure cygnus --help
Usage: tour-guide configure cygnus [-h | --help] [-u <username> | --mysql-user
<username>]
                                   [-p <password> | --mysql-password <password>]

Apply configuration changes for cygnus.

Command options:

  -h  --help                        Show this help.
  -u  --mysql-user <username>       Set the MySQL database user to use.
                                    Default value is 'root'.
  -p  --mysql-password <password>   Set the MySQL database password to use.
                                    Default value is 'mysql'.
```

i.e. to change the MySQL credentials used for Cygnus to `myuser` and `mypassword`, issue the following command:

```
$ ./tour-guide configure cygnus --mysql-user myuser --mysql-password mypassword
```

This will update the `docker-compose.yml` file and set the following variables for the Cygnus container:

```
- CYGNUS_MYSQL_USER=myuser
- CYGNUS_MYSQL_PASS=mypassword
```

This variables will be used by the Cygnus container to update the following MySQL credentials on the `agent.conf` configuration file when the container starts:

```
cygnus-ngsi.sinks.mysql-sink.mysql_username = myuser
cygnus-ngsi.sinks.mysql-sink.mysql_password = mypassword
```

For more information on other available configuration options for the Cygnus container, see Using the image on the Cygnus repository. For more complex configuration options, see the official documentation for Cygnus.

Besides the Cygnus container modifications, the following variables will be set for the MySQL container:

```
- MYSQL_USER=myuser
- MYSQL_PASSWORD=mypassword
```

```
- MYSQL_DATABASE=tourguide
```

The `MYSQL_DATABASE` variable defines the database to be used. As we are using a user with no superuser privileges, we need to tell the MySQL container to create the database for us. The database name will be the same as the 'Fiware Service' used for the sensors, which defaults to `tourguide` for the Tour Guide Application. If we were using the `root` user, there would be no need to specify the database name, as that would be created on demand.

Subscriptions

After the system is configured and running, we need to tell Orion that we want it to send the data as it changes. To do this we need to register a subscription on Orion for Cygnus.

Imagine we want to store the temperature changes on a restaurant over a period of time. We then need to tell Orion to send us the information whenever any of the temperature sensors value for the restaurant changes. To do this we need the following information:

- Id of the restaurant, e.g. `0115206c51f60b48b77e4c937835795c33bb953f`
- the type of the sensor, e.g. `temperature`
- the room of the sensor, e.g. `kitchen`

With this information, we will send a POST request to `http://localhost:1026/v1/subscribeContext` with a payload like this:

```
{
    "entities": [{
        "type": "Restaurant",
        "isPattern": "false",
        "id": "0115206c51f60b48b77e4c937835795c33bb953f"
    }],
    "attributes": ["temperature:kitchen"],
    "reference": "http://cygnus:5050/notify",
    "duration": "P1M",
    "notifyConditions": [{
        "type": "ONCHANGE",
        "condValues": ["temperature:kitchen"]
    }],
    "throttling": "PT1S"
}
```

The `reference` field specifies the Cygnus endpoint to use. The default used in the container is `http://cygnus:5050/notify`. Now, every time the value of the sensor changes, Orion will send a notification to Cygnus with this change and Cygnus will store it on the database. In this example we are just requesting changes for a single sensor in a single restaurant. If you want something more complex, or for a detailed description on subscriptions, see `Context subscriptions` under Context management using NGSI10 on the official documentation for Orion Context Broker.

Included with TourGuide, there are two sample subscription scripts available at docker/cygnus/subscriptions:

- subscription-therm-sensors.sh
- subscription-humidity-sensors.sh

Please note that to use these scripts you need to update your hosts file with the running containers (see `tourguide configure hosts --help` command) before executing them.

Once we have done the subscription, we will receive an initial notification with the current data. This can be seen in the log of the Cygnus container:

```
$ docker logs Cygnus
```

You should see something like this:

```
time=2016-09-28T13:40:46.872Z | ... | srv=tourguide | subsrv=/Franchise1 | comp=cygnus-
ngsi | op=getEvents | msg=com.telefonica.iot.cygnus.handlers.NGSIRestHandler[264] :
Received data ({  "subscriptionId" : "57ebc85e0698bea0a46bc2b1",  "originator" :
"localhost",  "contextResponses" : [    {   "contextElement" : {        "type" :
"Restaurant",      "isPattern" : "false",      "id" :
"0115206c51f60b48b77e4c937835795c33bb953f",      "attributes" : [        {
"name" : "temperature:kitchen",         "type" : "Number",          "value" : "25"
}       ]   },      "statusCode" : {        "code" : "200",     "reasonPhrase" : "OK"   }
}  ]})

...

time=2016-09-28T13:40:50.930Z | ... | srv=tourguide | subsrv=/Franchise1 | comp=cygnus-
ngsi | op=processNewBatches | msg=com.telefonica.iot.cygnus.sinks.NGSISink[417] : Batch
completed, persisting it
time=2016-09-28T13:40:50.931Z | ... | srv=tourguide | subsrv=/Franchise1 | comp=cygnus-
ngsi | op=persistAggregation | msg=com.telefonica.iot.cygnus.sinks.NGSIMySQLSink[455] :
[mysql-sink] Persisting data at OrionMySQLSink. Database (tourguide), Table
(Franchise1_0115206c51f60b48b77e4c937835795c33bb953f_Restaurant), Fields
((recvTimeTs,recvTime,fiwareServicePath,entityId,entityType,attrName,attrType,attrValue,a
ttrMd)), Values (('1475070046874','2016-09-
28T13:40:46.874','/Franchise1','0115206c51f60b48b77e4c937835795c33bb953f','Restaurant','t
emperature:kitchen','Number','25','[]'))
```

There we can see that Cygnus should have stored the information on the `Franchise1_0115206c51f60b48b77e4c937835795c33bb953f_Restaurant` table of the `tourguide` database. We can check if this is true by connecting to the MySQL database:

```
$ docker exec -i -t mysql mysql -u root -p tourguide

mysql> show tables;
+---------------------------------------------------------------+
| Tables_in_tourguide                                           |
+---------------------------------------------------------------+
| Franchise1_0115206c51f60b48b77e4c937835795c33bb953f_Restaurant |
+---------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> select * from Franchise1_0115206c51f60b48b77e4c937835795c33bb953f_Restaurant;
+---------------+-------------------------+-------------------+--------------------------
----------------+------------+---------------------+----------+-----------+--------+
| recvTimeTs    | recvTime                | fiwareServicePath | entityId
| entityType | attrName         | attrType | attrValue | attrMd |
+---------------+-------------------------+-------------------+--------------------------
----------------+------------+---------------------+----------+-----------+--------+
| 1475070046874 | 2016-09-28T13:40:46.874 | /Franchise1     |
0115206c51f60b48b77e4c937835795c33bb953f | Restaurant | temperature:kitchen | Number   |
25       | []    |
+---------------+-------------------------+-------------------+--------------------------
----------------+------------+---------------------+----------+-----------+--------+
1 row in set (0.00 sec)
```

As we can see, the temperature in the kitchen of the restaurant with Id `0115206c51f60b48b77e4c937835795c33bb953f` is `25` degrees. Let's see what happens when the temperature changes. To do this, we will use the following command:

```
$ ./tour-guide sensors send-data -i "0115206c51f60b48b77e4c937835795c33bb953f-kitchen-
temperature" -d "t|21"
```

We can see that Orion sends a new notification with the temperature change:

```
time=2016-09-28T14:01:42.136Z | ... | srv=tourguide | subsrv=/Franchise1 | comp=cygnus-
ngsi | op=getEvents | msg=com.telefonica.iot.cygnus.handlers.NGSIRestHandler[264] :
Received data ({  "subscriptionId" : "57ebc85e0698bea0a46bc2b1",  "originator" :
"localhost",  "contextResponses" : [    {   "contextElement" : {        "type" :
"Restaurant",      "isPattern" : "false",      "id" :
"0115206c51f60b48b77e4c937835795c33bb953f",      "attributes" : [        {
"name" : "temperature:kitchen",         "type" : "Number",          "value" : "21"
}       ]   },      "statusCode" : {        "code" : "200",     "reasonPhrase" : "OK"   }
}  ]})
time=2016-09-28T14:01:42.141Z | ... | srv=tourguide | subsrv=/Franchise1 | comp=cygnus-
ngsi | op=processNewBatches | msg=com.telefonica.iot.cygnus.sinks.NGSISink[363] : Batch
accumulation time reached, the batch will be processed as it is
time=2016-09-28T14:01:42.142Z | ... | srv=tourguide | subsrv=/Franchise1 | comp=cygnus-
ngsi | op=processNewBatches | msg=com.telefonica.iot.cygnus.sinks.NGSISink[417] : Batch
completed, persisting it
```

If we check the database again, we see the new value has been stored alongside the old value:

```
$ docker exec -i -t mysql mysql -u root -p tourguide

mysql> select * from Franchise1_0115206c51f60b48b77e4c937835795c33bb953f_Restaurant;
+---------------+-----------------------+---------------------+----------+-----------+--------+
---------------+------------+---------------------+----------+-----------+--------+
| recvTimeTs    | recvTime              | fiwareServicePath | entityId
| entityType | attrName         | attrType | attrValue | attrMd |
+---------------+-----------------------+---------------------+----------+-----------+--------+
---------------+------------+---------------------+----------+-----------+--------+
| 1475070046874 | 2016-09-28T13:40:46.874 | /Franchise1     |
0115206c51f60b48b77e4c937835795c33bb953f | Restaurant | temperature:kitchen | Number   |
25       | []    |
| 1475071302136 | 2016-09-28T14:01:42.136 | /Franchise1     |
0115206c51f60b48b77e4c937835795c33bb953f | Restaurant | temperature:kitchen | Number   |
21       | []    |
+---------------+-----------------------+---------------------+----------+-----------+--------+
---------------+------------+---------------------+----------+-----------+--------+
2 rows in set (0.00 sec)
```

We can check that this value is the one stored on Orion with:

```
$ curl --header 'Fiware-Service: tourguide'--header 'Accept: text/plain'
http://localhost:1026/v2/entities/0115206c51f60b48b77e4c937835795c33bb953f/attrs/temperat
ure:kitchen/value
```

This should return:

```
"21"
```

If the temperature keeps changing, the new values will be notified to Cygnus and stored in the database.

3.6 Using the front-end application

3.6.1 Running the front-end application

For running the front-end application, assuming you are on the folder where you downloaded the source code, follow these steps:

- Launch the application using the command line interface (from now on CLI) with: `./tour-guide start`
- Provide user, roles and permissions on Keyrock with `tour-guide configure keyrock`
- Get the Oauth credentials from Keyrock and add them to the Tourguide configuration with `./tour-guide configure oauth`
- Configure the hosts file in order to run the images using their aliases with `./tour-guide configure hosts`

After performing all the above steps, the front-end application is ready to be used. If you open `http://tourguide/client` in your web browser, you should see something like:

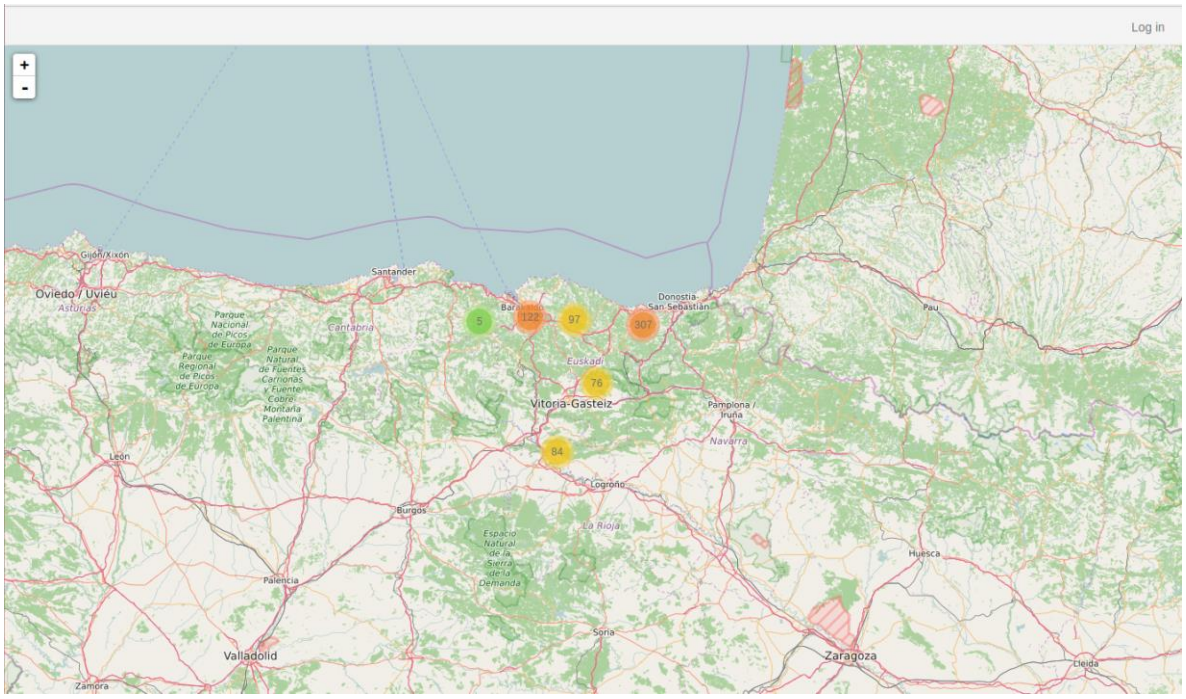

Figure 3 .- Front-end application main view.

You can log in using, for instance, the user `user1@test.com` with the password `test`. Once logged in, you can see a new menu at the top of the page:

- **Home**: Return to the main view where all restaurants are displayed.
- **My franchises** (only available for users that belong to a franchise): Display information filtered by user franchises. The sub menu entries available for each franchise are:

- **Restaurants**: Filter restaurants shown in the map by the selected franchise.
- **Reviews**: Display all reviews written about the selected franchise.
- **Reservations**: Display all reservations of the selected franchise.

- **My reservations**: Display a list with all the reservations made by the logged user.
- **My reviews**: Display a list with all reviews written by the logged user.

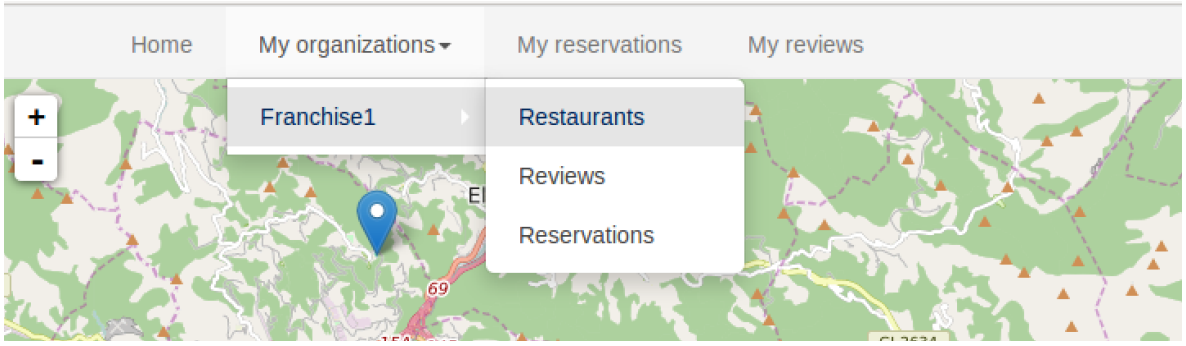

Figure 4 .- Front-end application menu.

3.6.1.1 *Front-end application main view*

In this section, all restaurants are represented in the map, in order to provide a clean view avoiding restaurant markers overlaps, they are grouped using a clustering approach. If you zoom in or click on a group, the restaurants or smaller groups are shown. Each group has a number that represents how many restaurants are collapsed. You need increase the zoom level or click on the groups until you get the desired restaurant. The following image shows a representation with restaurant marks and groups:

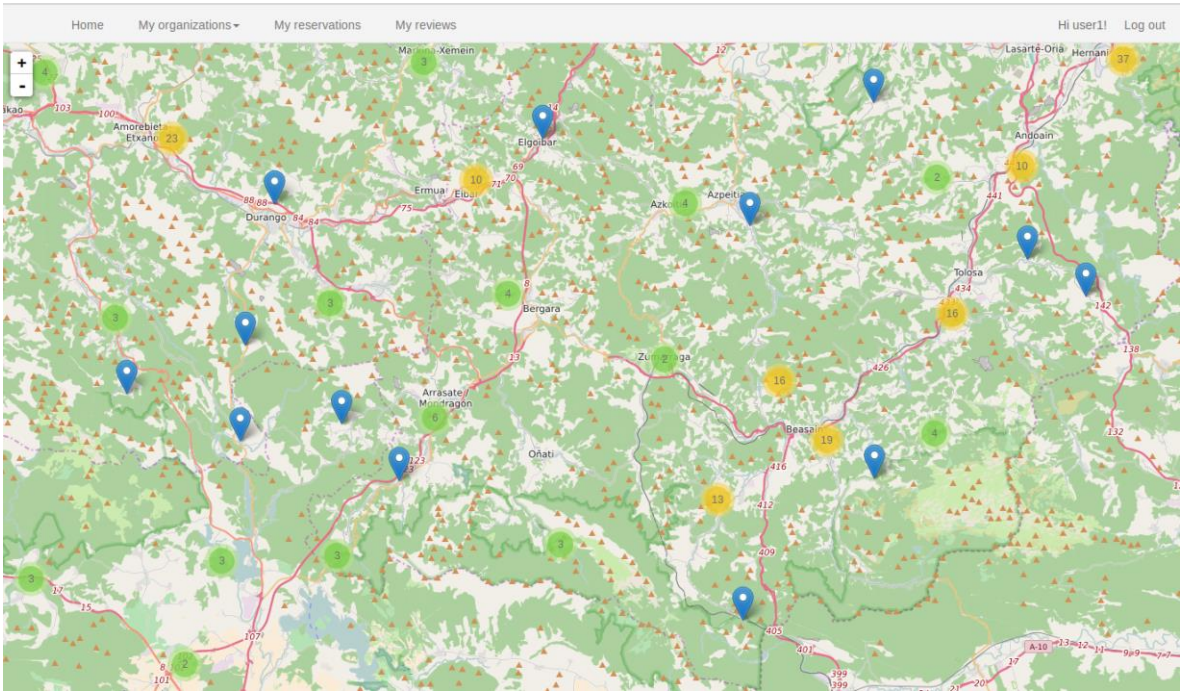

Figure 5 .- Example of front-end application main view with restaurant marks and grouped restaurants.

If a restaurant mark is clicked, a pop-up with its general information and some operations is shown:

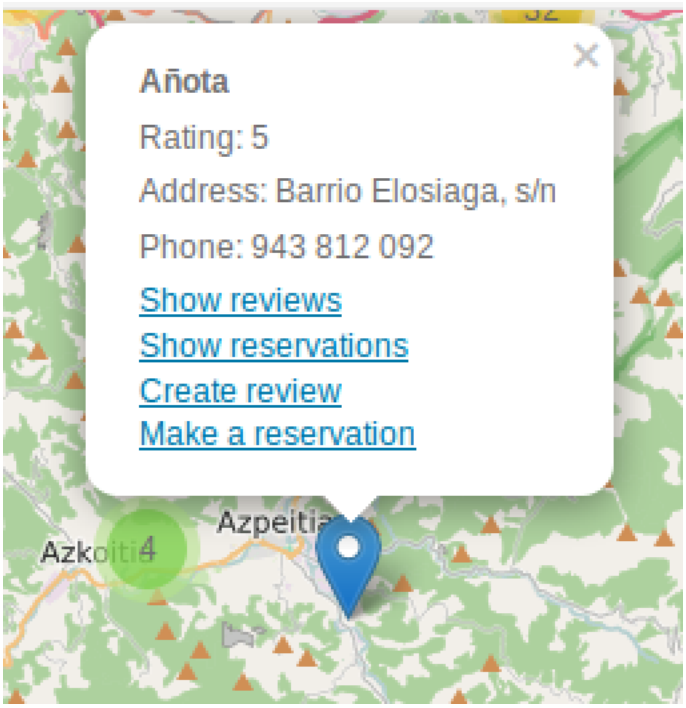

Figure 6.- Restaurant pop-up example.

"Show reviews" and "Show reservations" options are always displayed. You need to be logged in to see the "Make a reservation" option and not to belong to any franchise in order to create a review. If any of this options are clicked, a modal window comes up displaying the requested information or providing the input mechanism to perform the operation.

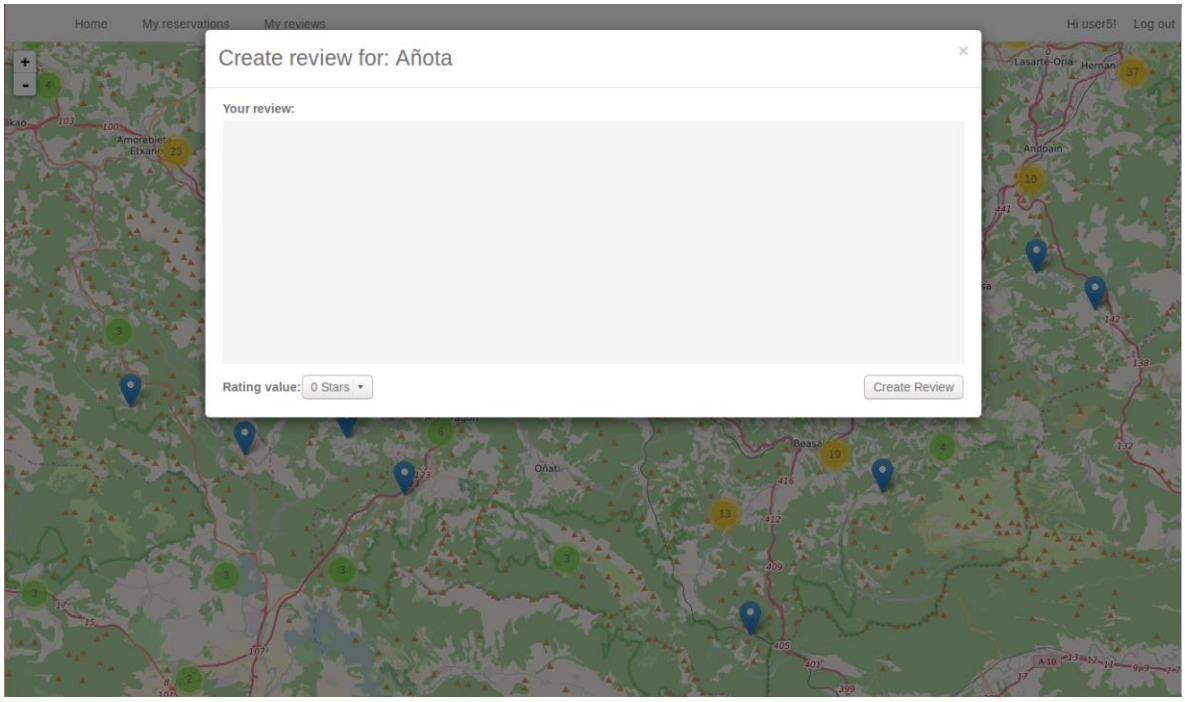

Figure 7 .- Example of review creation.

3.6.1.2 *Reviews filtered by user or by franchise*

Unlike how restaurant reviews are shown, reviews filtered by user or restaurant are displayed using a specific section.

All reviews are listed as depicted in the following image:

Restaurant	Rating			
Amelibia	5	View review	Edit review	Delete review
Virgen del Valle	1	View review	Edit review	Delete review
Lamioga	1	View review	Edit review	Delete review
Lasturko Taberna	2	View review	Edit review	Delete review
Calbeton	1	View review	Edit review	Delete review
Huri - Barrena	4	View review	Edit review	Delete review
Katxarro	5	View review	Edit review	Delete review
Begotxu	2	View review	Edit review	Delete review
Txabarri	1	View review	Edit review	Delete review
Taberna Berri	4	View review	Edit review	Delete review
Convento San Roque	3	View review	Edit review	Delete review
Via Fora	2	View review	Edit review	Delete review
San Bartolomé	1	View review	Edit review	Delete review
Tartalo	1	View review	Edit review	Delete review
Orbela	2	View review	Edit review	Delete review
Ysios	2	View review	Edit review	Delete review
Ordo-Zelai	1	View review	Edit review	Delete review
Olarizu	5	View review	Edit review	Delete review
Solar de Samaniego	2	View review	Edit review	Delete review

Figure 8 .- Reviews filtered by user.

“Edit review” and “Delete review” options are only available in “My reviews section”.

3.6.1.3 *Reservations filtered by user or by organization*

This case is similar to the reviews case, while restaurant reservations are shown as pop-ups in the main view, reviews filtered by the logged user or by organization are listed in a specific section:

Restaurant	Date	Diners	
Jauregia	23/11/2015 2:22:10	3	Cancel reservation
Arlobi	14/11/2015 17:14:00	7	Cancel reservation
Dolomiti	22/11/2015 22:11:50	7	Cancel reservation
La Bodeguilla	20/11/2015 4:09:47	10	Cancel reservation
Txoko Mendibile	9/11/2015 13:48:16	3	Cancel reservation
Faustino	20/11/2015 21:40:58	2	Cancel reservation
Aspaldiko	24/11/2015 15:06:33	1	Cancel reservation
Calbeton	11/11/2015 0:28:34	4	Cancel reservation
De Luis R.	9/11/2015 11:50:16	8	Cancel reservation
Balintxarreketa	5/11/2015 1:36:12	8	Cancel reservation
El Cartero	18/11/2015 0:55:00	5	Cancel reservation
Hambroneta	14/11/2015 3:40:50	1	Cancel reservation
Zalbide	23/11/2015 21:35:22	4	Cancel reservation
El Caserón	23/11/2015 18:19:29	3	Cancel reservation
Arandia	18/11/2015 6:36:18	5	Cancel reservation
Deslorian	15/11/2015 16:27:16	7	Cancel reservation
El Chalet	2/11/2015 6:35:20	3	Cancel reservation
Trujal Almazara de la Rioja Alavesa	15/11/2015 18:02:32	8	Cancel reservation
Zuia Plaza Kafe	3/11/2015 2:28:34	4	Cancel reservation

Figure 9 .- Reservations filtered by user.

3.6.2 Front-end application structure

The front-end application is built using HTML5, CSS and JS. These are the most important modules:

- **restaurantsAPI.js**: This module performs requests against the server API.
- **connectionsAPI.js**: This module contains the functions related to the user session.
- **drawModule.js**: This module contains the methods to build the UI.
- **clientLogic.js**: This module specifies the application behavior. It also interconnects the other modules.
- **leaflet.js**: JS library for rendering the interactive map.

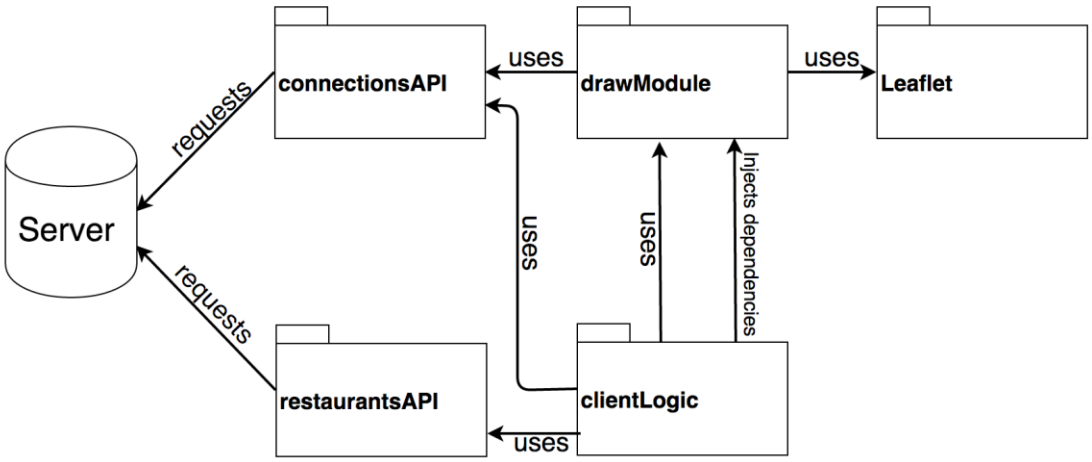

Figure 10 .- Structure of the front-end application main modules.

3.6.2.1 *restaurantsAPI.js*

This module is responsible for performing most of the requests between the front-end application and the server API. In its internal configuration the API endpoint, where all requests will be sent, is defined:

```
var baseURL = 'http://tourguide/api/orion/';
```

All functions provided by this module (except `simplifyRestaurantsFormat`) accept two callback functions, one to process the result data in case of success and another to process possible errors. For example, if `getAllRestaurants` is called as following:

```
getAllRestaurants(func1, func2);
```

If the API request responds with `200 OK`, the `func1` is called using the response as input parameter. However, if the API gets an error, for example a `404 NOT FOUND`, the `func2` is called instead.

The functions provided by this module are:

- `getAllRestaurants(sucessCallback, errorCallback)`: This function is used for retrieving all restaurants.
- `getOrganizationRestaurants(organization,sucessCallback, errorCallback)`: This function retrieves restaurants like `getAllRestaurants` but filtered by organization.
- `getOrganizationReviews(organization, sucessCallback, errorCallback)`: Retrieves all reviews written about all the restaurants of the specified organization.
- `getOrganizationRservations(organization, sucessCallback, errorCallback)`: Retrieves all reservations of the specified organization.

- `getRestaurantReviews(restaurantName, sucessCallback, errorCallback)`: Retrieves all reviews about the specified restaurant.
- `getRestaurantReservations(restaurantName, sucessCallback, errorCallback)`: Retrieves all reservations of the specified restaurant.
- `getUserReviews(username, sucessCallback, errorCallback)`: Retrieves all reviews written by the specified used.
- `getUserReservations(username, sucessCallback, errorCallback)`: Retrieves all reservations made by the specified user..
- `getReview(reviewID, sucessCallback, errorCallback)`: Retrieves a specific review.
- `createNewReview(restaurantName, ratingValue, reviewBody, sucessCallback, errorCallback)`: Creates a new review for the specified restaurant using the passed rating value and review content.
- `updateReview(reviewId, ratingValue, reviewBody, sucessCallback, errorCallback)`: Updates the specified review using the new provided values.
- `deleteReview(reviewId, sucessCallback, errorCallback)`: Deletes the specified review.
- `createNewReservation(restaurantName, partySize, reservationDatetime, sucessCallback, errorCallback)`: Creates a new reservation for the logged user at the specified restaurant and at the given date.
- `cancelReservation(reservationId, sucessCallback, errorCallback)`: Cancels the specified reservation.
- `simplifyRestaurantsFormat (restaurants)`: Takes an array of restaurants and return another one with the restaurants simplified. The expected format for the input array is the same than the format used for `getOrganizationRestaurants` or `getAllRestaurants`. This function also filters out restaurants with invalid coordinates.

3.6.2.2 *connectionsAPI.js*

This module manages the user information and the top menu. It has information about roles and permissions. The module provides the following functionalities:

- `loginNeeded(action)`: Executes the specified function only if the user is logged in.
- `loggedIn(userInfo)`: Stores the user information in the web browser `local storage` and create the top menu using the user information.
- `notLoggedIn()`: Removes the user information from the web browser `local storage` if it is recorded and create the top menu with the **log in** option.

- `hasRole(userInfo, roleName)`: Returns true if the specified user has the specified role.
- `getUser()`: Function that returns the information about the logged user.
- `roles`: object with the available roles.

This module also executes an init function that makes a query to the server API in order to know the user If such request is successful, the init function executes the `loggedIn` function creating the menu and storing the user information. Otherwise, it executes `notLoggedIn`, removing all information all previously stored user data and creating the menu.

3.6.2.3 *drawModule.js*

This module performs most of the operations strictly related to the UI. To use this module, you need to add the Leaflet, jQuery, jQueryUI and timepicker libraries. Unlike the other modules, this one needs to be initialized with the actions that will be performed when an event is triggered. This actions can be set up using the following functions:

- `setViewReservationAction(action)`
- `setViewRestaurantReviewsAction(action)`
- `setCreateNewReviewAction(action)`
- `setCreateNewReservationAction(action)`
- `setGetReservationsByDateAction(action)`
- `setViewReviewAction(action)`
- `setShowEditReviewAction(action)`
- `setUpdateReviewAction(action)`
- `setCancelReservationAction(action)`
- `setDeleteReviewAction(action)`

The functionalities provided by this module are:

- `addRestaurantstoMap(restaurants)`: This function adds a mark for each input restaurant to the Leaflet map. Each mark contains the relevant information about a restaurant and the available actions that users can perform over the restaurant.
- `setPopupTitle(title)`: Sets the title of the modal window.
- `setPopupContent(contentDiv)`: Replaces the modal window content with the specified div.
- `createReviewsDiv(reviews)`: Creates an element containing a list with the information of each input review.

- `createReservationsDiv(reservations)`: Creates an element containing a list with the information of each reservation.
- `createReviewsTable(reviews)`: Creates a table containing the basic informat information of each review. It also generates the **View review**, **Delete review** and **Update review** options.
- `createOrganizationReviewsTable(reviews)`: It is like createReviewsTable but it does not create the **Delete review** and **Update review** options. Besides, the reviews are filtered by organization.
- `createReservationsTable(reservations)`: Creates a table containing the information of each input reservation. It also generates the **Cancel reservation** option.
- `createReviewForm(restaurantName, review)`: Generates a form that creates a review for the specified restaurant. If review is passed, the form is prepared to be initialized with its values and form action is set to update review instead of create review.
- `initializeReviewForm(review)`: Initializes a review form that was previously created by `createReviewForm`.
- `createViewReviewDiv(review)`: Creates a div containing all the information about a review.
- `openPopUpWindow()`: Opens the modal window.
- `closePopUpWindow()`: Closes the modal window.

3.6.2.4 *clientLogic.js*

This module contains the client application behavior. It uses the `connectionsAPI` module for retrieving the user information, the `restaurantsAPI` module for performing requests against the server API and the `drawModule` for rendering the UI. This module also injects dependencies to the `drawModule`. This allows `drawModule` to know which actions should be executed from the UI.

This module contains the following methods:

- `showAllRestaurants()`: Performs a request for retrieving all restaurants and render them into the Leaflet map.
- `showOrganizationRestaurants(organization)`: Performs a request for retrieving all restaurants of an organization and render them in the leaflet map.
- `showReservationsByOrganization(organization)`: Performs a request retrieving all the restaurant reservations of the organization and displays them using a list.
- `showRestaurantReviews(name)`: Retrieves all reviews of the specified restaurant and displays them using the modal window.

- `showRestaurantReservations(name)`: Retrieves all reservations of the specified restaurant and displays them using the modal window.
- `showReviewsByUser(username)`: Retrieves all the reviews written for the specific user and creates a table displaying them.
- `showReviewsByOrganization(organization)`: Gets all the reviews of the organization restaurants and displays them.
- `createNewReview(name, rating, description)`: Creates a new review for the specified restaurant.
- `createNewReservation(name, partySize, time)`: Creates a new review for the specified restaurant.
- `updateReview(reviewId, rating, description)`: Updates the specified review.
- `deleteReview(reviewId)`: Deletes the specified the review.
- `showReservationsByUser(username)`: Retrieves all the reservations of the specified user and display them using a table.
- `getMyReviews()`: Calsl `showReviewsByUser` using the logged user information.
- `getMyReservations()`: Calls `showReservationsByUser` using the logged user information.
- `setUpDrawModule()`: Initializes all the actions to be performed from the UI.

3.6.3 How the front-end application works

All pages of the front-end application include the `connectionsAPI` module. This module executes an init function that tries to retrieve the user information performing a request against the server API endpoint `tourguide/client/user`. If the request is successful, the user information is stored for further operations and the top menu is created. If the user is not logged in, the request will fail, all user information previously stored is deleted and the login menu is generated.

In order to generate the top menu when the user is logged in, the `connectionsAPI` checks the user roles and the user organizations. For example, **My organizations** menu entry is only generated if the user belongs to at least one organization.

```
if (organizations.length > 0) {
    var myOrganizationsLi = document.createElement('LI');
    myOrganizationsLi.className = 'dropdown';
```

The init file of the main view sets up the Leaflet map centering the view in Vitoria, setting the zoom level to 8 and adding the Open Street Map layer:

```
map = L.map('map').setView([42.90816007196054, -
2.52960205078125], 8); // set tile layer
L.tileLayer('http://{s}.tile.osm.org/{z}/{x}/{y}.png', {
```

```
    attribution: '&copy; <a href="http://osm.org/copyright">OpenStreetMap</a> contributor
s'
}).addTo(map);
```

After the map creation. the init files initializes the `drawModule` and displays all restaurants:

```
clientLogic.setUpDrawModule();
clientLogic.showAllRestaurants();
```

The `clientLogic.showAllRestaurants` function uses `restaurantsAPI.getAllRestaurants` to retrieve the restaurants and `drawModule.addRestaurantstoMap` to add the restaurants to the map.

`drawModule.addRestaurantstoMap` iterates over the restaurants and creates a mark with a pop-up for each of them. The mark contains basic information about the restaurants like its address, telephone number, average rating, etc. The mark also displays available actions for the restaurant, these actions are binded using the methods that were injected by `clientLogic.setUpDrawModule()`. So, for example, if the **Show reviews** option is clicked the `clientLogic.showRestaurantReviews` is called. The following image explains the processes involved, which are executed in order from top to bottom.

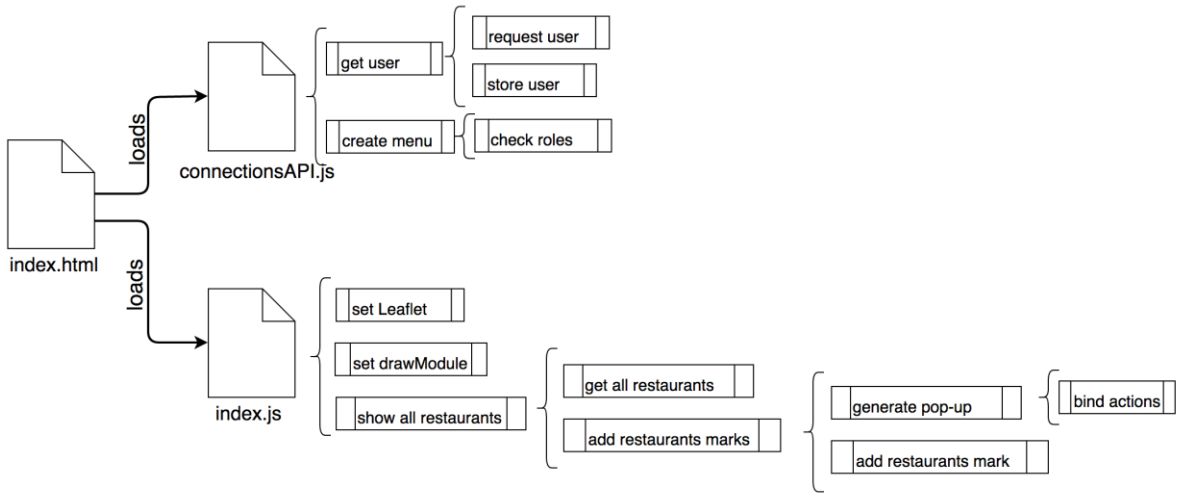

Figure 11 .- Processes involved for showing all restaurants.

My reviews section displays reviews written by the logged user. In order to check that the user is logged in the `connectionsAPI.loginNeeded` function is called with `clientLogic.getMyReviews()` as callback method. In other words, if the user is logged in the `clientLogic.getMyReviews()` is executed and if the user isn't, an error message requesting login is shown.

```
// only gets reviews if the user is logged
connectionsAPI.loginNeeded(function() {
    clientLogic.getMyReviews();
});
```

The `clientLogic.getMyReviews()` function uses `connectionsAPI.getUser()` to get the username, and then displays the user reviews using `clientLogic.showReviewsByUser`. Similarly to `restaurantsAPI.getAllRestaurants`, this function retrieves the user reviews using `restaurantsAPIgetUserReviews` and creates the reviews table using `drawModule.createReviewsTable`.

The rest of the sections follows the same structure as **main view** and **My reviews** sections.

4 Conclusions and future work

The FIWARE Tour Guide and its accompanying reference tutorial application have made a considerable step forward aimed at engaging even more developers in FIWARE. The project has worked intensively during this period, and as a result, below there is a description of the most remarkable achievements:

- Further alignment of the Tour Guide with the work produced by the technical chapters, by means of bidirectional feedback. Particularly the Tour Guide Application is using exactly the same Docker descriptors as the official ones released by the GEris.
- The structure of the Tour Guide Application has been improved, so that now it is more friendly for learning purposes. This has implied a refactoring to avoid automated procedures and substitute the former by manual procedures that will help learners to grasp what is happening behind the scenes
- A step by step guide has been produced in order to ease the learning process, following a try and tweak schema. This is complementary to the previous point.
- Improvements in the alignment between the examples of the Tour Guide and the Tour Guide Application data, so that learners can experiment more easily and conveniently

Although the Tour Guide Application is now in a fairly mature state, more work is planned to be conducted by the FIWARE Community during the following months:

- Use the Tour Guide application data as part of the API Cookbook of GEris. Particularly, this has already been successfully done with the FIWARE NGSIv2 cookbook.
- In cooperation with the FIWARE Cloud Chapter, deploy an instance of the Tour Guide Application in the FIWARE Lab using the new Docker container service developed under such chapter.
- Integrate more FIWARE GEris, for instance SpagoBI for data analytics and Kurento for stream processing.

All such future work is being tracked at https://github.com/Fiware/tutorials.TourGuide-App/issues together with Jira, so that the activities are properly planned as part of FIWARE's Agile process.