MONICA

Management Of Networked IoT Wearables – Very Large Scale Demonstration of Cultural Societal Applications
(Grant Agreement No 732350)

D7.5 The MONICA Development Toolbox 1

Date: 2018-12-31

Version 1.0

Published by the MONICA Consortium

Dissemination Level: Public

Co-funded by the European Union’s Horizon 2020 Framework Programme for Research and Innovation under Grant Agreement No 732350

Document control page

Document file:	D7.5 The MONICA Development Toolbox 1.docx
Document version:	1.0
Document owner:	CNet
Work package:	WP7 – Components & Cloud Integration
Task:	T7.6 – The MONICA Development Toolbox
Deliverable type:	[OTHER]
Document status:	Approved by the document owner for internal review Approved for submission to the EC

Document history:

Version	Author(s)	Date	Summary of changes made
0.1	Peeter Kool (CNet)	2018-10-18	Created ToC
0.2	Peter Rosengren	2018-11-30	Updated IoT Monitoring draft tool
0.3	ISMB	2018-12-05	SCRAL chapter
0.4	Mathias Axling (CNet)	2018-12-07	Updated Messager Broker
0.5	Peter Rosengren	2018-12-10	Added Service Catalogue
0.6	Peeter Kool	2018-12-11	Added Entity Catalogue
0.7	Vivian Esquivias (CNet)	2018-12-12	Reviewed chapter on IoT Monitoring Tool
0.9	Peter Rosengren, Peeter Kool	2019-01-05	Ready for peer review
1.0	Peter Rosengren, Peeter Kool	2019-01-20	Updated after peer review. Final version submitted to the European Commission

Internal review history:

Reviewed by	Date	Summary of comments
Nathalie Frey/Arjen Schoneveld, DEXELS	2019-02-15	Approved with comments
Pierre-Yves Hors, Optinvent	2019-02-15	Approved with comments

Index:

1. Executive Summary (CNet)

The MONICA Development Toolbox can be used to integrate various resources into the IoT Platform and hides the complexity of the communication with IoT devices. It features model-driven development of services that use the MONICA platform, also in connection with available Open Data sources. It will be based on a structure of service ontologies where a conceptual domain model describes the application, the services to be deployed and the objects involved (devices, users, rules, repositories, etc). The MONICA Toolbox is divided into three different categories:

- Software Developer Tools. These are packaged tools with user interfaces intended to be used by developers. They have been developed by the MONICA project.
- Generic Enablers. These are re-usable software components developed by the MONICA project. MONICA generic enablers are made available in an Open Source GIT repository and can be used by entrepreneurs, start-up and established companies alike.
- Third Party Services and Tools. These are some openly available third-party tools that are recommended by the MONICA project to use when building Large Scale IoT applications. MONICA Tools and Generic Enablers have available interfaces for these third-party tools and services.

2. Introduction

The MONICA project aims to provide a very large scale demonstration of multiple existing and new Internet of Things technologies for Smart Events. The solution will be deployed in 6 major cities in Europe. MONICA demonstrates a large scale IoT ecosystem that uses innovative wearable and portable IoT sensors and actuators with closed-loop back-end services integrated into an interoperable, cloud-based platform capable of offering a multitude of simultaneous, targeted applications.

This deliverable is based on work in task 7.5. The task develops an open software development toolbox and generic enablers that allow developers to rapidly develop new applications to be deployed on the MONICA platform. The development platform consists of a toolbox and a set of tutorials and guidelines.

The MONICA Toolbox is divided into three different categories:

- Software Developer Tools. These are packaged tools with user interfaces intended to be used by developers. They have been developed by the MONICA project. They are described in Chapter 3.
- Generic Enablers. These are re-usable software components developed by the MONICA project. MONICA generic enablers are made available in an Open Source GIT repository and can be used by entrepreneurs, start-up and established companies alike.
- Third Party Services and Tools. These are some openly available third-party tools that are recommended by the MONICA project to use when building Large Scale IoT applications. MONICA Tools and Generic Enablers have available interfaces for these third-party tools and services. They are described in Chapter 5.

The toolbox can be used to integrate various resources into the IoT Platform and hides the complexity of the communication with IoT devices. It features model-driven development of services that use the MONICA platform, also in connection with available Open Data sources. It will be based on a structure of service ontologies where a conceptual domain model describes the application, the services to be deployed and the objects involved (devices, users, rules, repositories, etc).

The MONICA Development Toolbox will be demonstrated in WP11 through the Entrepreneurship and Innovation programme where funding will be available for start-ups wishing to develop apps using the MONICA IoT toolbox and enablers. Deployment of the toolbox has been coordinated with the call for tenders for new apps in WP11 and the availability of demonstrations across the pilots.

This deliverable documents the first version of the MONICA Toolbox.

3. MONICA Software Developer Tools

3.1. IoT Monitoring Tool

In any IoT system there is always a need to keep track of the status of sensors and devices. Two main issues need to be considered – values that are not being reported and values that are erroneous. The IoT Monitoring Tool allows an operator to view the current status of the different deployed sensors and devices.

The main functionalities provided by the IoT Monitoring Tool are:

- Listing and overview of available sensors and devices
- Storage and presentation of meta data related to sensors/devices
    - Type of device, serial number, installation date, vendor
- Storage and presentation of meta data related to the physical location of the sensor/device
    - GPS coordinates or building information (room, floor), physical address, customer/owner data
- Meta data related to operation of the sensors
    - Expected reporting frequency, expected value ranges
- Presentation of current values and historical values
    - Tables and graphs
- Presentation of error states
    - Last reported values
- Maintenance related data
    - Last repair, frequency of errors/malfunction
    - Rights to sensor data, granularity level.

The IoT Monitoring tool provides access to the following data sources:

- Current Status Database
    - This represent the latest values of all available sensors, or the status flags in devices. It is stored in a SQL (Structured Query Language) database
- Historical Sensors Readings
    - OGC (Open Geospatial Consortium) compliant data store containing the data streams for the different sensors. Most likely NoSQL database
- IoT meta data
    - IoT-A compliant database to represent the IoT sensors and their mapping to the physical world being observed
- Domain Database
    - This contains the application specific data.

The IoT Display presented in D3.6 "IoT Display for Large Scale IoT Installations" has been largely implemented using the IoT Monitoring Tool.

3.2. IoT Entity, Service and Resource Catalogues

The Entity, Service and Resource Catalogues manage and store meta data related to the Entities being managed and monitored in the Large Scale Event and the available Services and Resources that can be used by applications. The catalogues also provide an? on boarding tool for populating the catalogues with relevant data. Furthermore it provides a query interface for retrieving data.

3.2.1. IoT Entity Catalogue

The IoT Entity and Resource Catalogue handles meta data and descriptions about the physical world entities that are under observation and actuation by the IoT world. The OGC SensorThings models the observation of Things using Sensors that produce DataStreams of Observations, see the OGC Model below. This needs to be complemented with a conceptual model for the application. The IoT Entity and Resource Catalogue stores and manages the relationship between these two models.

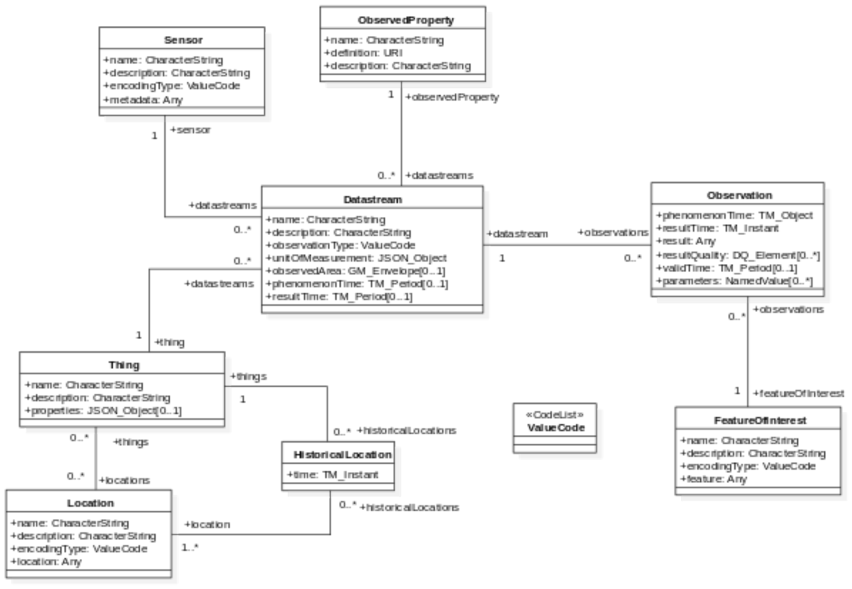


Figure 1 The OGC SensorThings Data Model

<u>OGC Registration API</u>

- Adding Things (including Locations)
    - Create, update and delete properties
    - Create, update and delete Datastreams for ObservedProperties
- Adding Sensors
- Adding FeatureOfInterest

<u>OGC Query API</u>

- Get Datastreams for Thing
    - Include Observations
- Get Datastreams for Sensor
    - Include Observations
- Get Observations for Datastream
    - Include FeatureOfInterest
- Get Things at Location
    - Include Datastreams
    - Include Datastreams and Observations

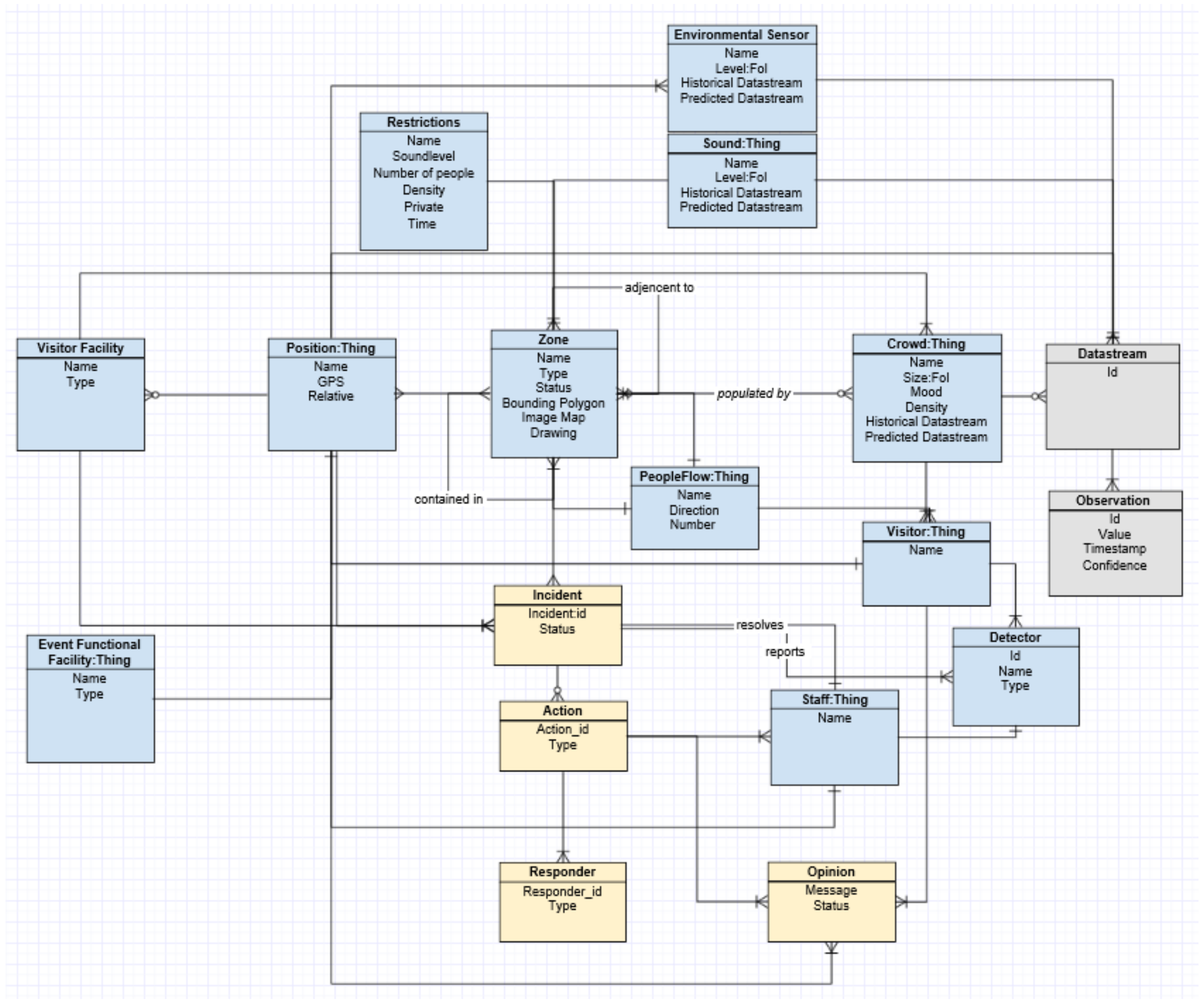


Figure 2 The IoT Entity Catalogue models the relationship between application model and OGC Sensorthings

<u>Entity Registration API</u>

- Adding Entity (with type)
  - Create, update and delete attributes
  - Create, update and delete
- Connect Entity and Thing
- Connect Entity attributes with Datastreams

<u>Entity Query API</u>

- Get Entities of Type
- Get Datastream Id for Entity attribute
- Get Entity Attribute
  - Include Observations

3.2.2. Service Catalogue

One of the challenges in MONICA is the large number of service instances due to the large number of pilot cases. There are event based services that generate messages and services that provide direct remote call interfaces. To install a new instance of MONICA requires configuration of all these services and their interfaces. The purpose of the IoT Service catalogue is to provide an easy way of configuring and maintaining the system.

This part of the catalogue is specifically addressing the services used in MONICA. The main functionalities of the MONICA IoT Service Catalogue:

- Provide discoverability of services
- Description of Services providing metadata
- Finding end points of services
- Finding which message topics are consumed by the service
- Finding which message topics are produced by the service
- Provide connection to the OGC SensorThings API model

The services are described by a Service Model which is derived from the Common Operational Picture Database, D6.4 “Decision Support Platform with Common Operational Picture 2”. It is possible to query about the existence of a service that provides certain information for instance ‘”crowd density”. Furthermore it is possible to get information about which endpoint the service resides and which are the events it produces. Also which MQTT topics to listen to in order to get relevant service information. Also the IoT catalogue will deliver information about which events the service consumes.

The Service Catalogue is based on the OGC SensorThings API Tasking model. This model extends the Sensing model of OGC to allow control and manipulation of the physical world objects in an IoT application.

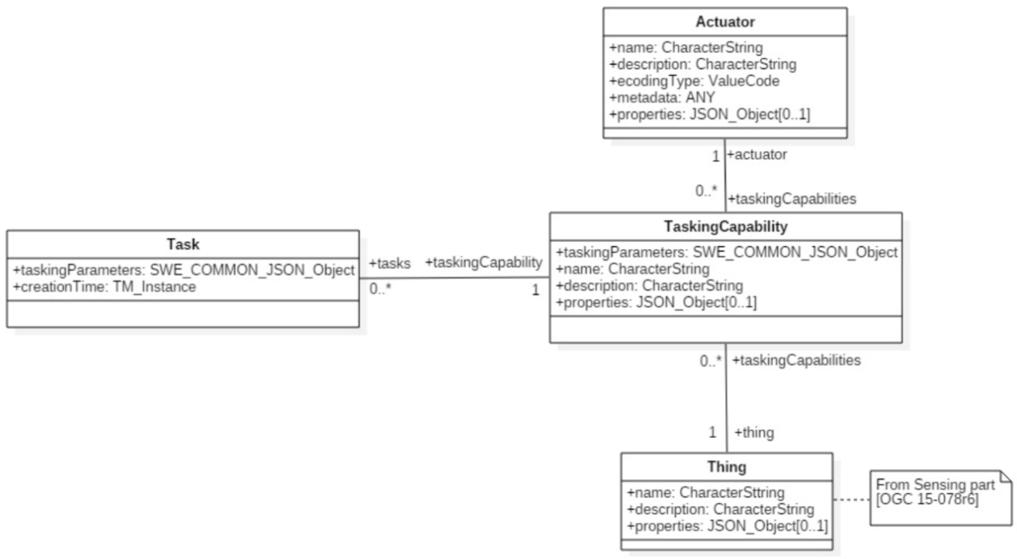

Actuator
+name: CharacterString
+description: CharacterString
+ecodingType: ValueCode
+metadata: ANY
+properties: JSON_Object[0..1]
1 +actuator
0..* +taskingCapabilities
TaskingCapability
+taskingParameters: SWE_COMMON_JSON_Object
+name: CharacterString
+description: CharacterString
+properties: JSON_Object[0..1]
Task
+taskingParameters: SWE_COMMON_JSON_Object
+creationTime: TM_Instance
+tasks +taskingCapability
0..* 1
0..* +taskingCapabilities
1 +thing
Thing
+name: CharacterSttring
+description: CharacterString
+properties: JSON_Object[0..1]
From Sensing part [OGC 15-078r6]

Figure 3 OGC Sensor things API Tasking Model

Thing

This is the same definition as in the Sensing part of OGC – “a thing is an object of the physical world (physical things) or the information world (virtual things) that is capable of being identified and integrated into communication networks”.

Actuator

An Actuator is a device that can be controlled/tasked. The Actuator entity contains information and metadata about a task-able actuator. Each TaskingCapability has one Actuator and defines the parameters that can be set/tasked for the Actuator.

TaskingCapability

The TaskingCapability entity contains information about the capabilities of the task-able device. It contains all the parameters that can be used for controlling the device. SWE Common JSON encoding rules [OGC17-011r2] are used to define these parameters for TaskingCapability.

Task

The Task entity contains the parameter detail of the control action that should be run on the task-able device.

The *TaskingCapabilities* are the services that an *Actuator* is offering. If an Actuator is a camera, the service would be “start/stop” and possibly “zoom”. The *Tasks* are the actual calls or instructions to carry out the service.

The Actuator could check the Task stream and when the Task “status:on” arrives it will actually carry out this action. However, this would probably be inefficient so it is also possible to subscribe and listen to when new tasks are created using MQTT and act upon this.

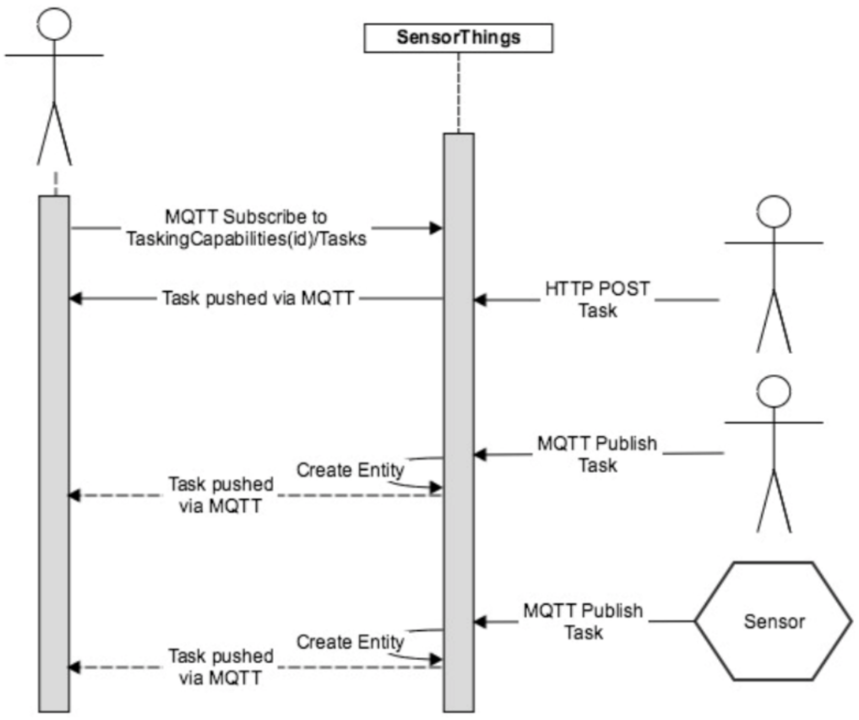


Figure 4 Listening to Task entities using MQTT

3.2.2.1. MONICA Tasking Model

Different MONICA Services can register themselves and describe their TaskingCapabilities using the OGC Tasking Model.

For instance a wearable wristband will register its Service “Display Text” as a TaskingCapability using the following JSON structure:

```
{
        "type": "Text",
        "name": "display",
        "label": "display text",
        "description": "Set text to display",
        "constraint": {
        "type": "AllowedTokens",
        "pattern": "^[a-zA-Z ]*$"
}
```

Other MONICA services or apps modules can then use this service by creating a Task like this:

```
{
        "display": "MONICA Alert. Crowded at exit Vesterbrogade"
}
```

The Wristband Actuator will subscribe to these Task and execute them when they arrive. In MONICA there are 5 types of services that can be invoked and used through the OGC Tasking model:

- Information Services (Display, Updates, Alerts)
- Instructional Services
- Configuration Services
- Action Services
- Subscription Services

Information Services
These are services for displaying a message to a user, visitor, operator etc. Typically an alert on a wristband or a pop up message in an app.

Instructional Services
This type of service is intended for guiding or instructing an operator or a security guard.

Configuration Service
This service allows configuration of an IoT Device or sensor, for instance configuring an algorithm for object detection in a camera.

Action Service
This type of service performs a state changing action in the physical world, for instance turning on a light or starting an alarm bell, reducing the sound level.

Subscription Service
This service allows to start a subscription to listen to messages from a Service or initiate a callback function.

3.2.2.2. IoT Service Catalogue Registration API

Register Service

```
{
"type": "Visitor Communication",
"name": "Visitor Wristband Display",
"description": "displays the given text on a wristband"
}
```

Service/TaskingCapabilities

Adds or updates a TaskingCapability of Service

```
{
"name": "alert",
"description": "displays the given text on a wristband",
"taskingParameters": {
        "type": "Text",
        "name": "display",
        "label": "display text",
        "description": "Set text to display",
        "constraint": {
                "type": "AllowedTokens",
                "pattern": "^[a-zA-Z ]*$"
                }
        "sender":  "id of service that generates the text message"
        }
        "properties": {
        }
}
```

3.2.2.3. IoT Service Catalogue Query API

- Search for Services by Function
- Search for Services by Location
- Search for Services by Message topics

3.3. IoT Resource Management and Catalogue Tool

SCRAL is the acronym for Smart City Resource Adaptation Layer and it is a kind of software wrapper that collects a set of drivers responsible for the integration of heterogeneous resources. Specifically, the SCRAL is in charge with the abstraction and virtualisation of physical entities within the MONICA IoT platform, so that specific devices and sub-systems functionalities are exposed to upper layers through a common interface which is agnostic to technology.

Among main physical resources management activities, there are three topics relevant to the SCRAL:

- Resources Configuration;
- Resources Discovery;
- Resources Health Monitoring.

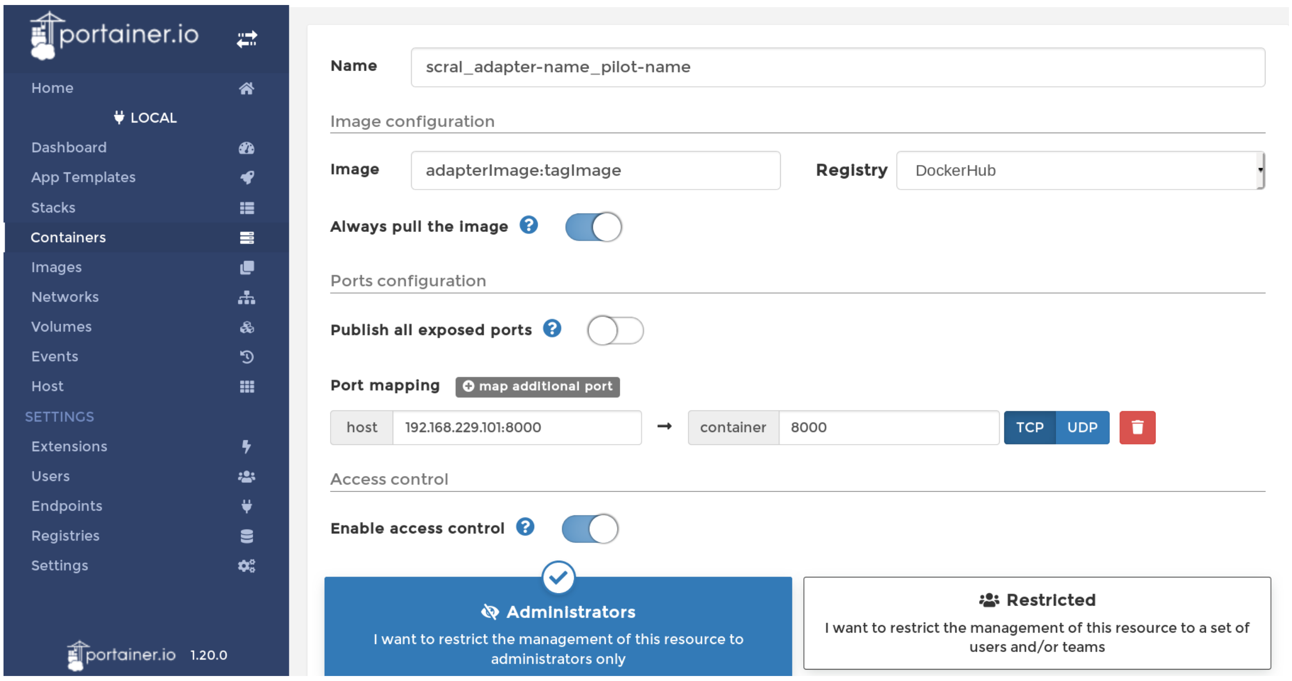
3.3.1. Resources Configuration

In general, the configuration process provides values for the correct behaviour of components that are instantiated for the realisation of some specific functionality. Within an adaptation context, features may address the configuration of both physical and virtual parameters.

In the first case, parameters for querying sensors can be set, including sensing data rate and publishing topic. To this purpose, the Control Core enabled by the SCRAL through the OGC modelling system can be used to reach specific device gateways and forward the information via REST and/or MQTT interface (more details are reported in MONICA D3.4).

On the other hand, configuration of virtual parameters refers to the way the integrated modules are exposed for external services. This task is accomplished by the SCRAL with a strong support from Portainer[1], a lightweight management user interface which allows to easily manage Docker containers, images, volumes and networks. SCRAL adapters are packaged in Docker images, pushed to the MONICA cloud and then deployed as pilot-specific configured containers. Figure 5 shows an example of the “Add container” interface used by the SCRAL to properly configure the adapter to be deployed. Container name, source image, port mapping and enable access control are the main configurable parameters for a container.

[1] https://www.portainer.io/#Home



Name: scral_adapter-name_pilot-name

Image configuration

Image: adapterImage:tagImage

Registry: DockerHub

Always pull the image: on

Ports configuration

Publish all exposed ports: off

Port mapping: map additional port

host 192.168.229.101:8000 → container 8000 TCP UDP

Access control

Enable access control: on

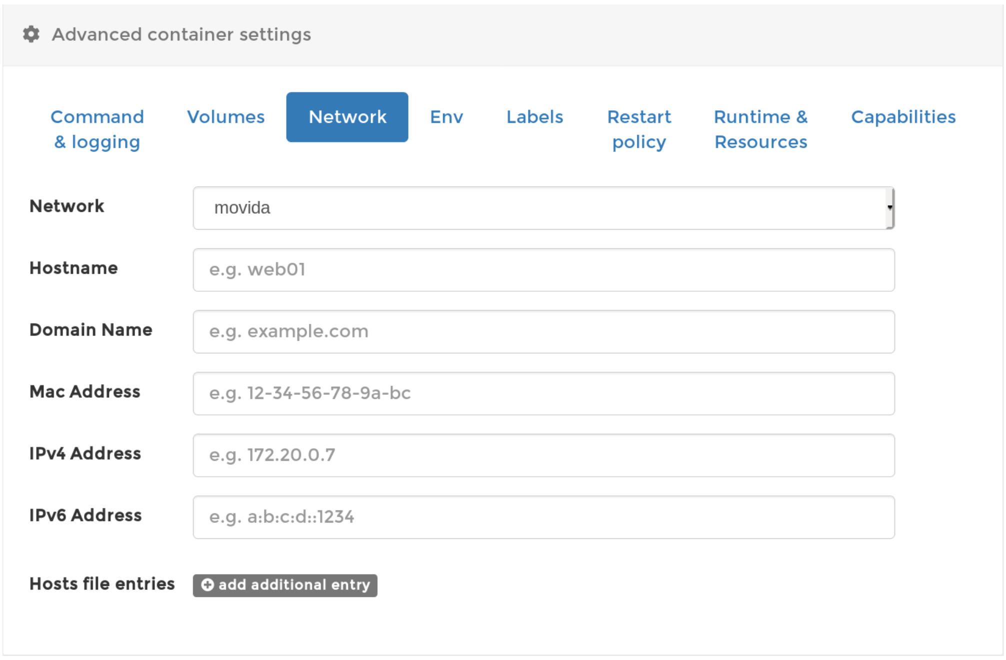
Administrators (selected): I want to restrict the management of this resource to administrators only

Restricted: I want to restrict the management of this resource to a set of users and/or teams

Figure 5 Deployment configuration for a SCRAL adapter container via Portainer tool

Furthermore, Figure 5 and Figure 7 show some of the available advanced settings that allow the assignment for the container to a virtual network as well as the restart behaviour in case the container would stop. As result, the whole set of the presented parameters allows the operator to import host images and easily manage a secure and functional deployment for different adapters running in parallel active pilots. In fact, the same integration module might be double instantiated within two different networks over two different listening endpoints, and separately linked with two different back end instances, so that each large scale installation would rely on its own IoT infrastructure.

In addition to resources configuration support, there are other useful features provided by Portainer about log monitoring and console access. More details will be described in section 3.3.3 Resources Health Monitoring.



Network: movida

Hostname: e.g. web01

Domain Name: e.g. example.com

Mac Address: e.g. 12-34-56-78-9a-bc

IPv4 Address: e.g. 172.20.0.7

IPv6 Address: e.g. a:b:c:d::1234

Hosts file entries: add additional entry

Figure 6 Advanced settings for a for a SCRAL adapter container via Portainer tool - Network configuration

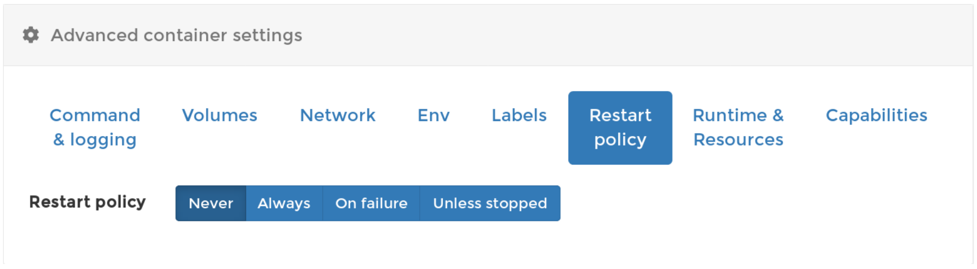


Figure 7 Advanced settings for a SCRAL adapter container via Portainer tool - Restart policy

3.3.2. Resources Discovery

With the discovery process it is possible to collect information about devices and modules registered to a platform. This information can be useful to have an overview of the status of the platform, and also to easily access device and services catalogues for monitoring purposes.

Because of the several technologies involved in the implementation of the physical layer, in most cases the SCRAL waits for subscription requests sent by the Edge layer to a specific listening endpoint. Each request registers a device into the MONICA platform, usually attached with static information relevant for the integration of the device itself, such as physical identifier, type of sensor, timestamp of the generated event etc. The SCRAL listening interface is always active for each adapter, so new resources can be added at runtime whenever needed.

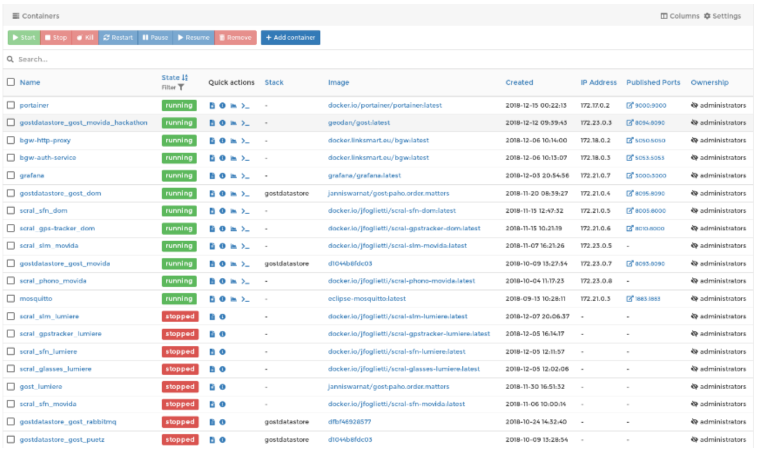
In other scenarios, where a push interface is not available from the physical layer (especially in the case of external platform integration), the SCRAL actively runs the discovery of available devices and stores the found information as for the other resources. In this case, once the integration process has started, if a new resource is added to the system, the SCRAL should run the discovery process again.

The collected data during the discovery process is internally stored and exposed on demand by the SCRAL integration module via REST and MQTT connectors. The most useful representation is given by the local resource catalogue, where for each deviceID-propertyName couple there is an OGC Datastream assigned. The mapping between physical resources and Datastream IDs is the key information for the application developers to know where to get the expected values within the OGC semantic framework (see MONICA D3.4 – Chapter 5 for details). Due to the complexity of the framework, the proper IDs might not always be straightforward to find. For this reason, the local resource catalogue could ease the IoT developers in the finding, and also helping them for the tasks of data subscription and retrieval. Other information from the physical layer might be represented as well, made available through the SCRAL and finally shown via visualisation tools.

3.3.3. Resources Health Monitoring

Checking the status of the resources is a task that can be applied at different levels of the platform. At the very low level, it might happen that sensors and gateways go offline for a period of time and events are not updated anymore. When issues originates from some network or back end failure, Portainer offers good monitoring solutions for deployed resource adapters, server and volumes.

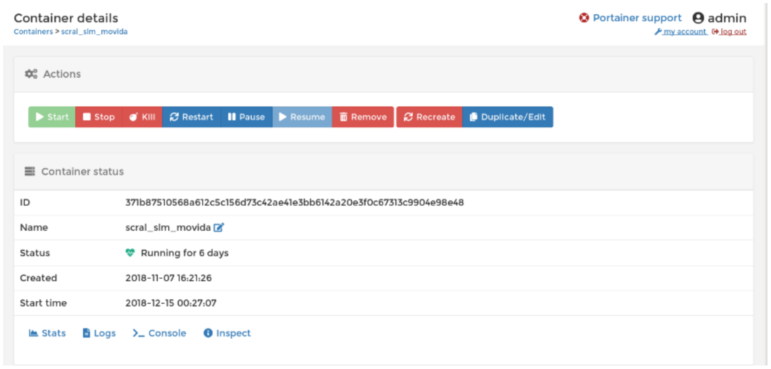
Figure 8 shows the Portainer interface where all the deployed containers are represented by name, status, endpoint and other context information.



Name	State Filter	Quick actions	Stack	Image	Created	IP Address	Published Ports	Ownership
portainer	running	🗎 ⓘ 📈 >_	-	docker.io/portainer/portainer:latest	2018-12-15 00:22:13	172.17.0.2	9000:9000	administrators
gostdatastore_gost_movida_hackathon	running	🗎 ⓘ 📈 >_	-	geodan/gost:latest	2018-12-12 09:39:43	172.23.0.3	8094:8090	administrators
bgw-http-proxy	running	🗎 ⓘ 📈 >_	-	docker.linksmart.eu/bgw:latest	2018-12-06 10:14:00	172.18.0.2	5050:5050	administrators
bgw-auth-service	running	🗎 ⓘ 📈 >_	-	docker.linksmart.eu/bgw:latest	2018-12-06 10:13:07	172.18.0.3	5053:5053	administrators
grafana	running	🗎 ⓘ 📈 >_	-	grafana/grafana:latest	2018-12-03 20:54:56	172.21.0.7	3000:3000	administrators
gostdatastore_gost_dom	running	🗎 ⓘ 📈 >_	gostdatastore	janniswarnat/gost:paho.order.matters	2018-11-20 08:39:27	172.21.0.4	8095:8090	administrators
scral_sfn_dom	running	🗎 ⓘ 📈 >_	-	docker.io/jfoglietti/scral-sfn-dom:latest	2018-11-15 12:47:32	172.21.0.5	8005:8000	administrators
scral_gps-tracker_dom	running	🗎 ⓘ 📈 >_	-	docker.io/jfoglietti/scral-gpstracker-dom:latest	2018-11-15 10:21:19	172.21.0.6	8010:8000	administrators
scral_slm_movida	running	🗎 ⓘ 📈 >_	-	docker.io/jfoglietti/scral-slm-movida:latest	2018-11-07 16:21:26	172.23.0.5	-	administrators
gostdatastore_gost_movida	running	🗎 ⓘ 📈 >_	gostdatastore	d1044b8fdc03	2018-10-09 13:27:54	172.23.0.7	8093:8090	administrators
scral_phono_movida	running	🗎 ⓘ 📈 >_	-	docker.io/jfoglietti/scral-phono-movida:latest	2018-10-04 11:17:23	172.23.0.8	-	administrators
mosquitto	running	🗎 ⓘ 📈 >_	-	eclipse-mosquitto:latest	2018-09-13 10:28:11	172.21.0.3	1883:1883	administrators
scral_slm_lumiere	stopped	🗎 ⓘ	-	docker.io/jfoglietti/scral-slm-lumiere:latest	2018-12-07 20:06:37	-	-	administrators
scral_gpstracker_lumiere	stopped	🗎 ⓘ	-	docker.io/jfoglietti/scral-gpstracker-lumiere:latest	2018-12-05 16:14:17	-	-	administrators
scral_sfn_lumiere	stopped	🗎 ⓘ	-	docker.io/jfoglietti/scral-sfn-lumiere:latest	2018-12-05 12:11:57	-	-	administrators
scral_glasses_lumiere	stopped	🗎 ⓘ	-	docker.io/jfoglietti/scral-glasses-lumiere:latest	2018-12-05 12:02:06	-	-	administrators
gost_lumiere	stopped	🗎 ⓘ	-	janniswarnat/gost:paho.order.matters	2018-11-30 16:51:32	-	-	administrators
scral_sfn_movida	stopped	🗎 ⓘ	-	docker.io/jfoglietti/scral-sfn-movida:latest	2018-11-06 10:00:14	-	-	administrators
gostdatastore_gost_rabbitmq	stopped	🗎 ⓘ	gostdatastore	dfbf46928577	2018-10-24 14:32:40	-	-	administrators
gostdatastore_gost_puetz	stopped	🗎 ⓘ	gostdatastore	d1044b8fdc03	2018-10-09 13:28:54	-	-	administrators

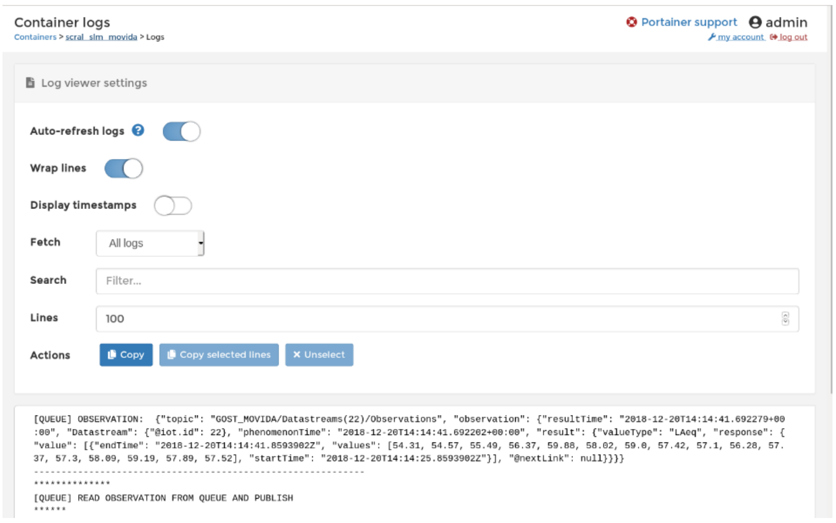
Figure 8 Container list and status information provided by Portainer

The "State" field is the first flag returning whether the container is properly running or not. Depending on the "running" or "stopped" conditions, the operator can either enter the log file for each container as well as the console of the container itself, as shown in Figure 9. The "Logs" session is a space that records the events that occur in the container and where errors, problems and warnings are constantly logged and saved for analysis (Figure 10). Log files are indeed always available, even though the container stopped for some reason.



ID	371b87510568a612c5c156d73c42ae41e3bb6142a20e3f0c67313c9904e98e48
Name	scral_slm_movida
Status	Running for 6 days
Created	2018-11-07 16:21:26
Start time	2018-12-15 00:27:07

Figure 9 Container details on Portainer



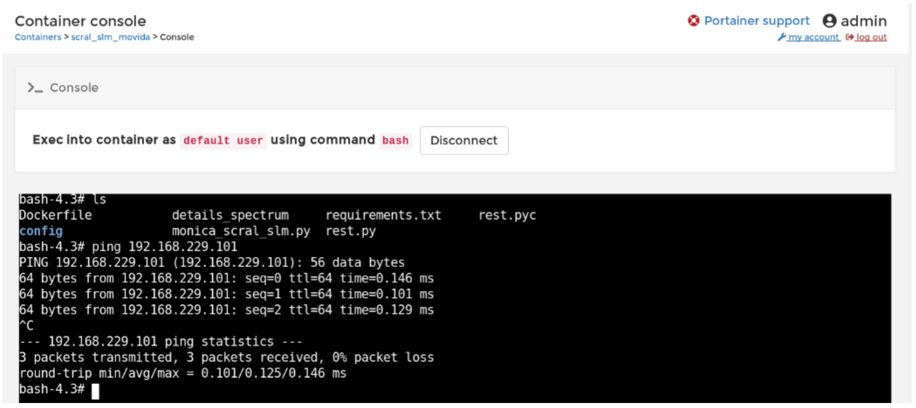
Auto-refresh logs	on
Wrap lines	on
Display timestamps	off
Fetch	All logs
Search	Filter...
Lines	100

```
[QUEUE] OBSERVATION:  {"topic": "GOST_MOVIDA/Datastreams(22)/Observations", "observation": {"resultTime": "2018-12-20T14:14:41.692279+00
:00", "Datastream": {"@iot.id": 22}, "phenomenonTime": "2018-12-20T14:14:41.692202+00:00", "result": {"valueType": "LAeq", "response": {
"value": [{"endTime": "2018-12-20T14:14:41.8593902Z", "values": [54.31, 54.57, 55.49, 56.37, 59.88, 58.02, 59.0, 57.42, 57.1, 56.28, 57.
37, 57.3, 58.09, 59.19, 57.89, 57.52], "startTime": "2018-12-20T14:14:25.8593902Z"}], "@nextLink": null}}}}
------------------------------------------------------------
**************
[QUEUE] READ OBSERVATION FROM QUEUE AND PUBLISH
******
```

Figure 10 Container logs on Portainer

However, for connectivity issues towards external components, container log files might be not sufficient to fully understand where the problem comes from. For this reason, each SCRAL integration module is “containerised” from a Docker image that supports bash access, so that, when the container is still running, it is possible to enter it via the “Console” bash interface and explore its internal status as well ping possibly unreachable modules, as depicted in Figure 11.

Moreover, other statics and performance analysis might be run either from the “Stats” section of “Container details” view or also from the bash console itself.



Exec into container as `default user` using command `bash`

```
bash-4.3# ls
Dockerfile          details_spectrum    requirements.txt    rest.pyc
config              monica_scral_slm.py  rest.py
bash-4.3# ping 192.168.229.101
PING 192.168.229.101 (192.168.229.101): 56 data bytes
64 bytes from 192.168.229.101: seq=0 ttl=64 time=0.146 ms
64 bytes from 192.168.229.101: seq=1 ttl=64 time=0.101 ms
64 bytes from 192.168.229.101: seq=2 ttl=64 time=0.129 ms
^C
--- 192.168.229.101 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.101/0.125/0.146 ms
bash-4.3# 
```

Figure 11 Container console on Portainer

4. Generic Enablers

The MONICA Generic Enablers will be made available on Git as on Open Source Repository. Pre-prepared Docker Containers will also be made available. This will be described in the second release of this deliverable.

5. Third-party Tools

5.1. Message and Event Broker Configuration

The MONICA project has selected RabbitMQ[2] as the implementation mechanism for the message broker. RabbitMQ is an open source component supplied under the Mozilla Public License. RabbitMQ is a widely used open source message broker[3] with an extensible architecture. It implements the AMQP 0-9-1 protocol[4] (Advanced Message Queuing Protocol) and can - through extension mechanisms, plugins - support the most common messaging protocols, e.g. MQTT (Message Queuing Telemetry Transport), STOMP (Streaming Text Oriented Messaging Protocol) and XMPP (Extensible Messaging and Presence Protocol). Extensions and adapters can be written to support other messaging patterns, protocols and security management solutions.

RabbitMQ implements AMQP 0-9-1 and the AMQP concepts of messages, producers, exchanges, queues and consumers. Each of these exists within an administrative unit called virtual host. A broker may contain several virtual hosts, and the users, defined in RabbitMQ, can be assigned with read, write and administrative rights by the host.

5.1.1. Basic Concepts

The basic concepts of a message broker is that *Producers* and *Consumers* communicate through *Messages*. Messages are managed or routed through *Exchanges*. Exchanges are basically a set of queues together with a principle for how the messages are distributed to these queues.

A publisher – an application that produces messages - sends a message to an exchange, in which it is routed to queues. Then the message is pushed to (or pulled by) a consumer – an application that processes messages - for processing. The producer, consumers and the broker can all reside on different brokers. The following section describes these basic concepts of AMQP 0.9.1 in the context of the RabbitMQ implementation.

5.1.1.1. Producers

A producer is an application that sends messages to an exchange. The producer may be any application written in any programming language, using an AMQP client API. The producer sets the attributes and the content of the message, including routing information, and sends the message to an exchange on a broker host. The producer specifies whether messages should be persisted or transient. Furthermore the producer specifies what should happen with messages that cannot be routed to a queue.

A producer in MONICA could be a sensor at the main gate reporting people passing, or an intelligent camera detecting a queue, or a wristband reporting a location.

5.1.1.2. Consumers

A consumer can be considered any application that receives messages from a queue and is identified by a consumer tag string. The messages can be delivered to the consumer by the AMQP push API or fetched by the consumer using the AMQP pull API. It is possible to register more than one consumer per queue or declare one consumer as the exclusive consumer for the queue. The consumer can send acknowledgement messages back to the host to indicate whether the message has been received or rejected.

Typical consumers in MONICA are the Common Operational Picture (Deliverable D6.4) and IoT Display (Deliverable D3.6)

5.1.1.3. Messages

An AMQP message consists of a header with attributes and application data. Attributes consist of key-value pairs. The properties consist of optional application-specific properties and a set of standard message

[2] https://www.rabbitmq.com/
[3] At the time of writing 35.000 production deployments , https://www.rabbitmq.com/
[4] http://www.amqp.org/sites/amqp.org/files/amqp0-9-1.zip

delivery annotations defined by the AMQP specification, e.g. message id, correlation id, time to live, delivery mode, priority, routing key and header dictionary.

The routing key or header dictionary are “addressing” attributes set by the producer to specify which queue(s) a message should be distributed by the exchange. The delivery mode attribute of a message can be declared persistent by the publisher – it is transient by default – and then it must be persisted between server restarts.

The application data is the actual content of the message. Actually it is a byte array which is not inspected by the broker. It is entirely application-specific and could be e.g. UTF-8 encoded text, XML, JSON, or Protocol Buffer byte format. AMQP defines an optional type system for specifying content and encoding type of the application data.

5.1.1.4. Queues

Queues are named first-in-first-out buffers in a message broker host that stores messages in memory or on disk. The messages are kept in the queue until a consumer is connected. Then the messages are delivered (in sequence) to the receiving application. The queues can be shared or private to a consumer. When a queue is private only the consumer knows about it and it will be removed once the consumer stops listening.

5.1.1.5. Bindings

A binding is the relation between a queue and an exchange and defines how messages should be routed from the exchange to the queue. Bindings are created or destroyed by applications over time to shape the message flow to queues. When a message arrives at the exchange the message attributes - routing key or header dictionary – set by the producer are evaluated to see if the binding has a match. If the binding matches, the message is copied to the queue. How the matching is done depends on the type of exchange.

5.1.1.6. Exchanges

Messages are sent from a producer to an exchange. Exchanges are defined per message broker host and they are responsible for routing the messages to queues. The way the messages are routed depends on the routing keys or headers set by the producer, the queue binding and the type of exchange.

The exchanges can be configured as durable, temporary or auto deleted when they are created. Durable exchanges will survive server restarts and will remain in the broker until explicitly deleted. Temporary exchanges exist until RabbitMQ is shut down. Auto deleted exchanges are deleted when the last producer or binding is removed from the exchange.

The dead letter exchange is an AMQP extension provided by RabbitMQ. The default exchange behaviour is to drop messages for which there is no binding providing a matching queue. The dead letter exchange will capture messages that cannot be delivered, which will be an important part of operational management of the system.

5.1.1.6.1. Direct exchange

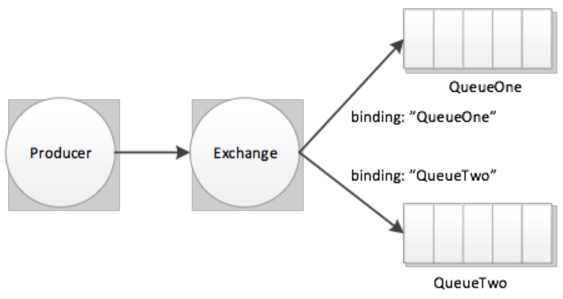


Figure 12 Direct exchange

A direct exchange delivers messages to queues based on the message routing key. A message is routed to the queues whose binding key has an exact match with the routing key of the message, e.g. a message with routing key “log” would be delivered to all queues with binding key “log”. A common practice is to use the queue name as routing key. If there is no matching binding, the message is discarded. AMQP specifies that an unnamed default exchange must be implemented which is a direct exchange. All queues must be bound to the unnamed exchange using the queue name as routing key.

Direct exchange is typically used when the producer has good knowledge about how the message should be used and by whom.

5.1.1.6.2. Fanout exchange

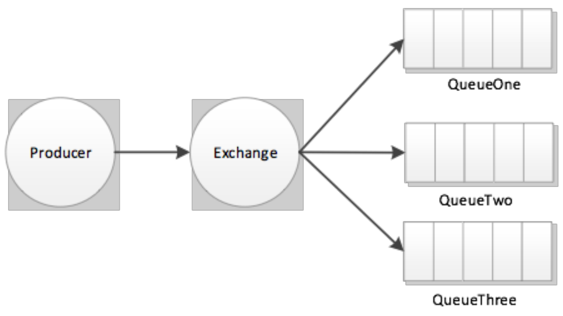


Figure 13 Fanout exchange

In a Fanout exchange, the messages are routed to all queues that are bound to the exchange. Any routing keys or headers are ignored. This is a useful pattern when broadcasting to several consumers that may process the message in different ways, e.g. logging, notification and aggregation.

A typical reason to use Fanout exchange is for messages that should be processed differently by different consumers. In the context of MONICA this could for instance be applied when a “violence situation” has been detected This message should be sent to Decision Support which will advice guards, it should also be processed by the Common Operational Picture and logged by the IoT Display.

5.1.1.6.2.1. Topic exchange

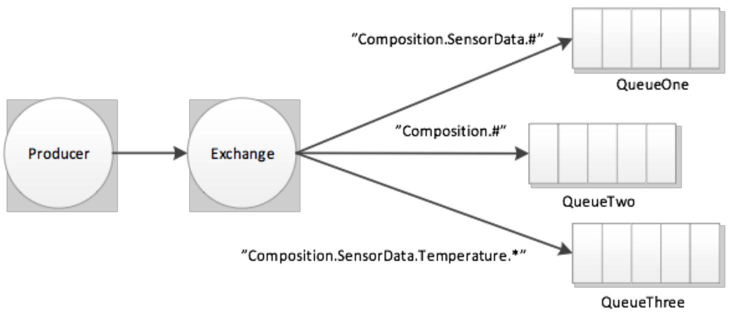

"Composition.SensorData.#"
"Composition.#"
"Composition.SensorData.Temperature.*"
Producer
Exchange
QueueOne
QueueTwo
QueueThree

Figure 14 Topic exchange

The topic exchange uses the routing key to distribute messages to queues. A topic routing key consists of zero or more words separated by dots “.”, e.g. “MONICA.MainGate.PeopleData”. The binding defines a routing pattern by the same rule, where “*” is used as a wildcard for a single word and “#” is used as a wildcard for zero or more words. If the binding for one or several queues matches the routing key, the message is distributed to these queues. This is very similar to the topic hierarchy and matching in MQTT (exchanging “/” for “.”).

A typical use for topic exchanges is to implement a publish-subscribe messaging pattern. In the context of MONICA this will correspond to sensors regularly reporting observations such as sound levels or air temperature.

5.1.1.6.3. Headers exchange

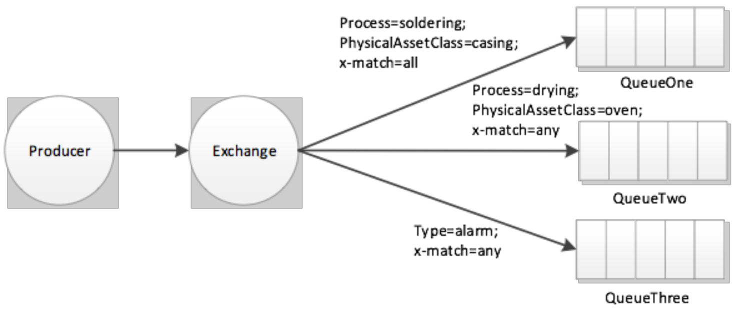
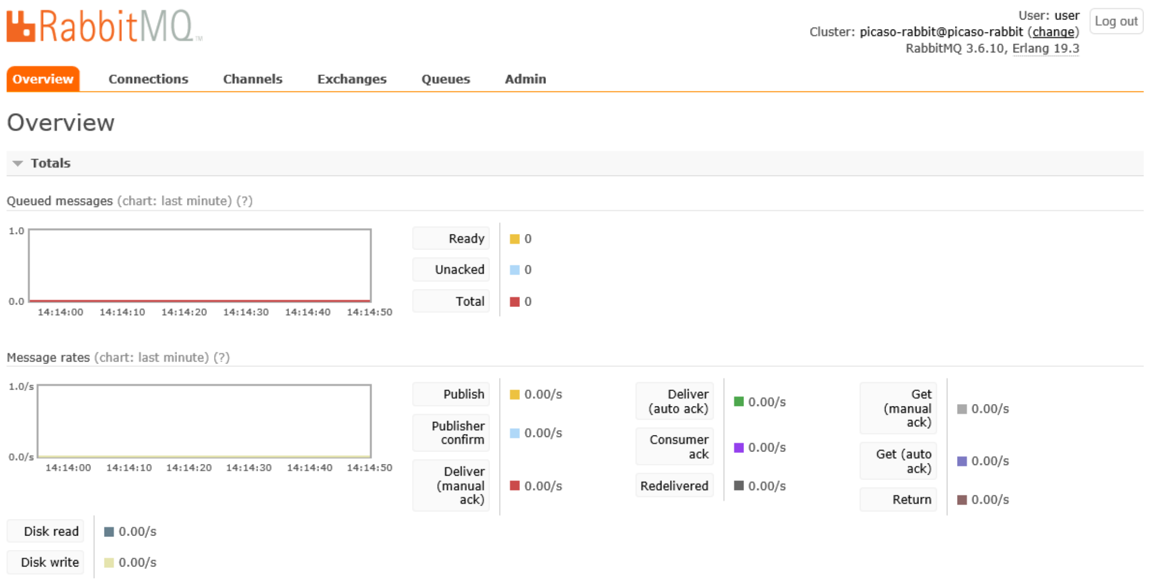


Figure 15 Headers exchange

The headers exchange allows for slightly more flexible routing than the topic exchanges. The routing key is not used in header exchanges. Instead, the message headers attribute, containing keys-value pairs, is used. The queue binding specifies the header keys to be matched and (optionally) the values that these should have. If the binding does not specify a value for the header key, it is sufficient for the key to be present in the message header for the binding key to match the message key. If the binding specifies a value for a key, the message header key must match this value. The binding attribute “x-match” specifies whether the logical “AND” or “OR” should be used when combining the matches of header binding keys. If “x-match=all” is specified, all key-value pairs in the binding must match the header for the message to be routed to that queue. The value “x-match=any” indicates that if any of the key-value pairs in the binding matches one or more in the message header, the message will be routed to that queue.

5.1.2. Message Broker Monitoring, Configuration and Management

Monitoring, configuration and management of RabbitMQ is handled by the RabbitMQ-management plugin. This plugin provides a command line tool, a browser-based UI and an HTTP-based API for monitoring and alerts that can be used to integrate with system operations management tools. Via the command line tool, set up, configuration and deployment of a Rabbit MQ server for a specific system can be scripted and automated using e.g. Docker files[5] and -images.



User: user
Cluster: picaso-rabbit@picaso-rabbit (change)
RabbitMQ 3.6.10, Erlang 19.3

Overview

Totals

Queued messages (chart: last minute) (?)

Ready	0
Unacked	0
Total	0

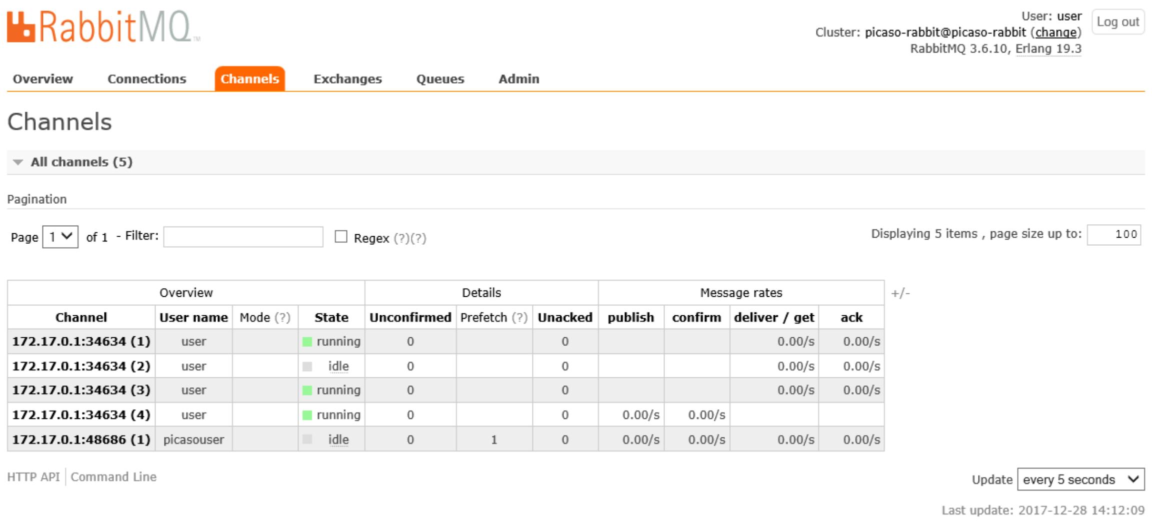
Message rates (chart: last minute) (?)

Publish	0.00/s	Deliver (auto ack)	0.00/s	Get (manual ack)	0.00/s
Publisher confirm	0.00/s	Consumer ack	0.00/s	Get (auto ack)	0.00/s
Deliver (manual ack)	0.00/s	Redelivered	0.00/s	Return	0.00/s

Disk read	0.00/s
Disk write	0.00/s

Figure 16 RabbitMQ Management Console

[5] https://docs.docker.com/engine/reference/builder/


RabbitMQ
User: user
Log out
Cluster: picaso-rabbit@picaso-rabbit (change)
RabbitMQ 3.6.10, Erlang 19.3
Overview
Connections
Channels
Exchanges
Queues
Admin

Channels

All channels (5)

Pagination

Page 1 of 1 - Filter: ☐ Regex (?)(?)

Displaying 5 items , page size up to: 100

Overview				Details			Message rates			
Channel	User name	Mode (?)	State	Unconfirmed	Prefetch (?)	Unacked	publish	confirm	deliver / get	ack
172.17.0.1:34634 (1)	user		🟩 running	0		0			0.00/s	0.00/s
172.17.0.1:34634 (2)	user		⬜ idle	0		0			0.00/s	0.00/s
172.17.0.1:34634 (3)	user		🟩 running	0		0			0.00/s	0.00/s
172.17.0.1:34634 (4)	user		🟩 running	0		0	0.00/s	0.00/s		
172.17.0.1:48686 (1)	picasouser		⬜ idle	0	1	0	0.00/s	0.00/s	0.00/s	0.00/s

HTTP API | Command Line

Update every 5 seconds

Last update: 2017-12-28 14:12:09

Figure 17 RabbitMQ Channels

The AMQP protocol is programmable; queues, exchanges and bindings are defined dynamically by the publishers and consumers. This means that for the most part, messaging patterns, topic schemes and routing topologies are defined by the applications using RabbitMQ, not by the broker administrators. However, performance and scalability settings (i.e. clusters, federations and the shovel) require setup by the broker administrators and are configured either from command line or from the management UI. E.g., a RabbitMQ cluster configuration may be set up for deployment by the Docker compose tool. User management, memory configuration and installed plugins (e.g., the MQTT or federation plugins) are also handled from the management plugin and web UI.

5.1.2.1. Monitoring the Message Broker with Nagios

Nagios is a widely-used free and open-source application for the monitoring of all kinds of IT systems. The Nagios server can be deployed on a dedicated machine or as part of a Docker environment and can monitor all kinds of local or remote infrastructure and services. It is extensible using plugins for specific monitoring requirements. The open source version of Nagios comes with a pre-defined set of plugins. Many more are available thanks to the open source community. Very specific monitoring requirements for any kind of IoT service could be met with custom-developed script in any programming language and integrated with Nagios as a user-defined plugin.

The Nagios server can be configured to notify administrators and other stakeholders if the system status is out of normal parameters. Notifications can be sent via e-mail or any kind of instant messaging system. Nagios also notifies the receivers if the service returns back to normal operation. It also provides a web GUI with configurable dashboards, graphs and reports.

There is a dedicated plugin available for the monitoring of RabbitMQ deployments[6]. This plugin queries the REST API of the RabbitMQ management interface to check e.g. service aliveness or number of active connections.

The more generic open source plugin check-mqtt[7] can be used to monitor the stream of MQTT messages sent to the broker. The plugin can monitor a list of topics for messages with certain keywords and produce a warning or error if no expected messages appear in a defined timeframe.

[6] https://github.com/nagios-plugins-rabbitmq/nagios-plugins-rabbitmq
[7] https://github.com/jpmens/check-mqtt

5.1.3. Management of offline situations

An important problem to manage is when sensors and gateways go offline for a period of time, especially sensors that report with high frequency. The most common reason is that the Internet connection is lost, but it could also be that the receiver is temporarily not functioning. There are several problems that could arise from this:

- Display of a phenomenon will lack data to render
- Applications that are taking decision based on time series and trends will all of a sudden lack data
- Learning algorithms will not be properly trained if a large amount of data is missing.

In the context of MONICA this could mean that the decision support components cannot properly assess a situation since they don't have the proper knowledge about what is the previous status at the event, such as how many people have passed through a gate, what have the sound levels been for the last hour.

The standard behaviour is that sensors/gateway cache the messages they are not able to send up to the limit of what their buffers allow. When they come online again they need to re-send the message to the message broker. Then there is a problem of over flooding the message broker and the corresponding storage facility. Therefor the re-sending needs to be controlled in a scheduled manner depending on the capacity of the receiver. Several schemes can be developed:

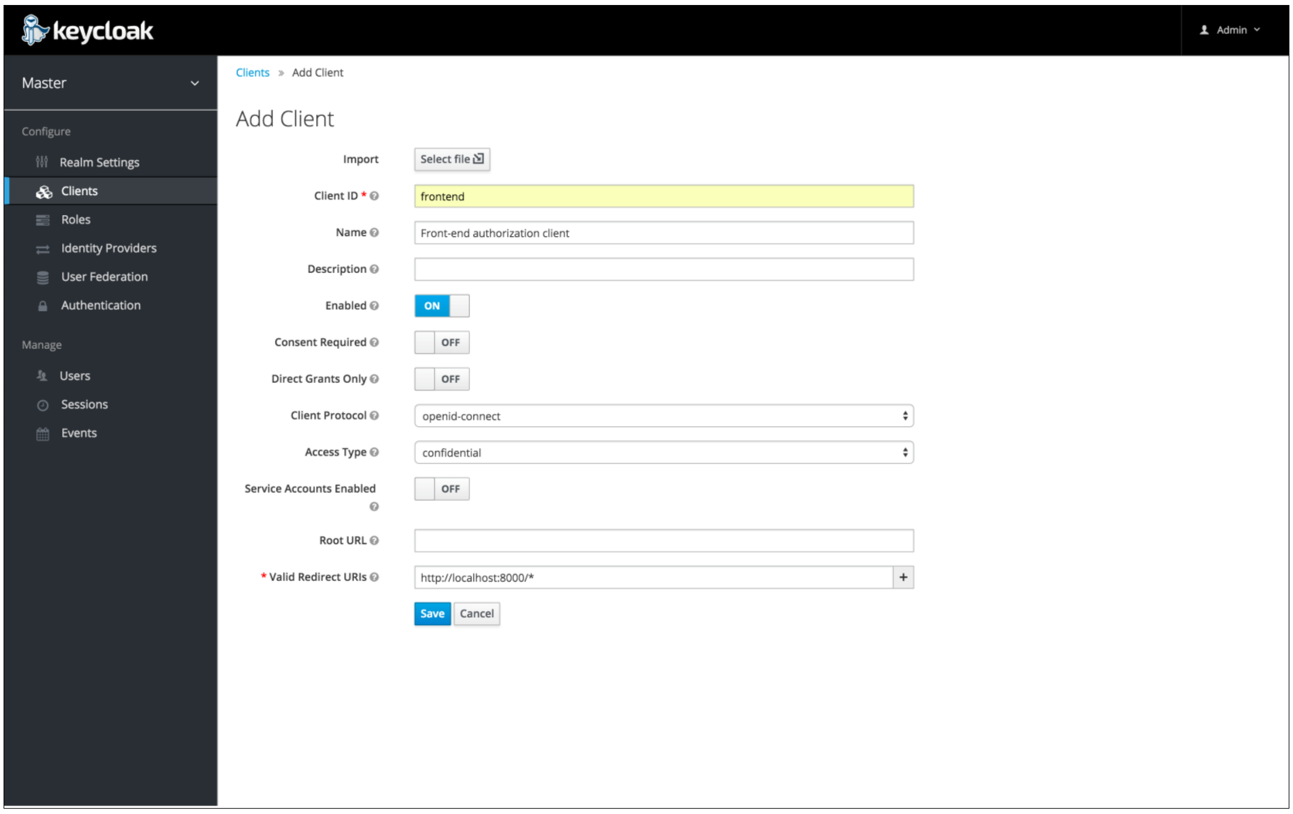
- Re-sending messages one- by-one in a timely manner
- Packaging many messages and send a compressed package.

5.2. KeyCloak

Keycloak[8] is an open source Identity and Access Management solution. Some of the features are:

- Single-Sign On: Authenticate on Keycloak rather on different applications. One single login will allow access to multiple applications and/or services
- Identity Brokering and Social Login: Enable login with social networks such as Google, Facebook, Twitter and GitHub
- User Federation: Connect directly to LDAP and Active Directory servers
- Standard Protocols: OpenID Connect OAuth 2.0 and SAML.

[8] http://www.keycloak.org



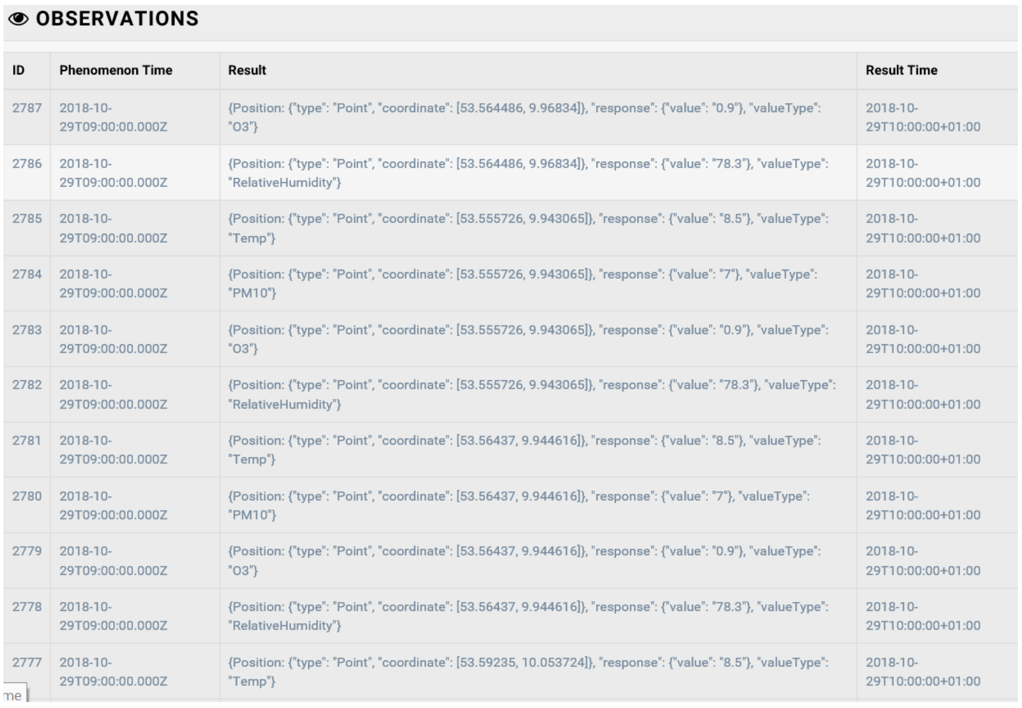
Add Client

Import	Select file
Client ID *	frontend
Name	Front-end authorization client
Description	
Enabled	ON
Consent Required	OFF
Direct Grants Only	OFF
Client Protocol	openid-connect
Access Type	confidential
Service Accounts Enabled	OFF
Root URL	
* Valid Redirect URIs	http://localhost:8000/*

Caption is missing.

5.3. GOST

GOST is a database based on a? Progress relational database that implements the OGC Sensorthings Data Model. GOST implements the OGC Sensorthings API and provides a simple dashboard that a developer can use to check stored data and make different listings of OGC data.



OBSERVATIONS

ID	Phenomenon Time	Result	Result Time
2787	2018-10-29T09:00:00.000Z	{Position: {"type": "Point", "coordinate": [53.564486, 9.96834]}, "response": {"value": "0.9"}, "valueType": "O3"}	2018-10-29T10:00:00+01:00
2786	2018-10-29T09:00:00.000Z	{Position: {"type": "Point", "coordinate": [53.564486, 9.96834]}, "response": {"value": "78.3"}, "valueType": "RelativeHumidity"}	2018-10-29T10:00:00+01:00
2785	2018-10-29T09:00:00.000Z	{Position: {"type": "Point", "coordinate": [53.555726, 9.943065]}, "response": {"value": "8.5"}, "valueType": "Temp"}	2018-10-29T10:00:00+01:00
2784	2018-10-29T09:00:00.000Z	{Position: {"type": "Point", "coordinate": [53.555726, 9.943065]}, "response": {"value": "7"}, "valueType": "PM10"}	2018-10-29T10:00:00+01:00
2783	2018-10-29T09:00:00.000Z	{Position: {"type": "Point", "coordinate": [53.555726, 9.943065]}, "response": {"value": "0.9"}, "valueType": "O3"}	2018-10-29T10:00:00+01:00
2782	2018-10-29T09:00:00.000Z	{Position: {"type": "Point", "coordinate": [53.555726, 9.943065]}, "response": {"value": "78.3"}, "valueType": "RelativeHumidity"}	2018-10-29T10:00:00+01:00
2781	2018-10-29T09:00:00.000Z	{Position: {"type": "Point", "coordinate": [53.56437, 9.944616]}, "response": {"value": "8.5"}, "valueType": "Temp"}	2018-10-29T10:00:00+01:00
2780	2018-10-29T09:00:00.000Z	{Position: {"type": "Point", "coordinate": [53.56437, 9.944616]}, "response": {"value": "7"}, "valueType": "PM10"}	2018-10-29T10:00:00+01:00
2779	2018-10-29T09:00:00.000Z	{Position: {"type": "Point", "coordinate": [53.56437, 9.944616]}, "response": {"value": "0.9"}, "valueType": "O3"}	2018-10-29T10:00:00+01:00
2778	2018-10-29T09:00:00.000Z	{Position: {"type": "Point", "coordinate": [53.56437, 9.944616]}, "response": {"value": "78.3"}, "valueType": "RelativeHumidity"}	2018-10-29T10:00:00+01:00
2777	2018-10-29T09:00:00.000Z	{Position: {"type": "Point", "coordinate": [53.59235, 10.053724]}, "response": {"value": "8.5"}, "valueType": "Temp"}	2018-10-29T10:00:00+01:00

Caption is missing

6. Conclusion

This deliverable has described the first version of the MONICA Development Toolbox.

The toolbox is currently divided into three different categories:

- Software Developer Tools. These are packaged tools with user interfaces intended to be used by developers. They have been developed by the MONICA project. So far the Software Developer Tools have proved efficient in allowing the project team to quickly create many different pilot use cases.
- Generic Enablers. These are re-usable software components developed by the MONICA project. They are intended to be integrated by developers in their applications.
- Third Party Services and Tools. These are some openly available third-party tools that are recommended by the MONICA project to use when building Large Scale IoT applications. MONICA Tools and Generic Enablers have available interfaces for these third-party tools and services. One conclusion to be drawn is that the use of Open standards and Open Source tools have facilitated a rapid development and deployment of the Monica platform.

The toolbox can be used to integrate various resources into the IoT Platform and hides the complexity of the communication with IoT devices. It features model-driven development of services that use the MONICA platform, also in connection with available Open Data sources.

7. List of Figures