BIROn - Birkbeck Institutional Research Online

Douzis, K. and Sotiriadis, Stelios and Petrakis, E.G.M. and Amza, C. (2018) Modular and generic IoT management on the cloud. Future Generation Computer Systems 78 (1), 369 - 378. ISSN 0167-739X.

Downloaded from: https://eprints.bbk.ac.uk/id/eprint/21786/

Usage Guidelines:
Please refer to usage guidelines at https://eprints.bbk.ac.uk/policies.html or alternatively contact lib-eprints@bbk.ac.uk.

Modular and Generic IoT Management on the Cloud

Konstantinos Douzis

Department of Electronic and Computer Engineering,
Technical University of Crete (TUC)
Chania, Crete, GR-73100, Greece
e-mail: kostasdouzis@gmail.com

Stelios Sotiriadis

Department of Electronic and Computer Engineering,
Technical University of Crete (TUC)
Chania, Crete, GR-73100, Greece
e-mail: s.sotiriadis@intelligence.tuc.gr

Euripides Petrakis

Department of Electronic and Computer Engineering,
Technical University of Crete (TUC)
Chania, Crete, GR-73100, Greece
e-mail: petrakis@intelligence.tuc.gr

Cristiana Amza

The Edward Rogers Sr. Department of Electrical and Computer Engineering,
University of Toronto, Bahen Centre for Information Technology
St. George Campus, 40, Toronto, ON M5S 2E4, Canada
e-mail: amza@ece.utoronto.ca

Abstract

Cloud computing and Internet of the things encompasses various physical devices that generate and exchange data with services promoting the integration between the physical world and computer-based systems. This work presents a novel future Internet cloud service for data collection from Internet of the things devices in an automatic, generalized and modular way. It includes a flexible API for managing devices, users and permissions by mapping data to users, publish and subscribe to context data as well as storage capabilities and data processing in the form of NoSQL big data. The service has been implemented in OpenStack platform and is easily deployable, reusable

and reconfigurable. The contributions of this work includes the on the fly data collection from devices that is stored in cloud scalable databases, the vendor agnostic Internet of things device connectivity (since it is designed to be flexible and to support device heterogeneity), and finally the modularity of the event based publish/subscribe service for context oriented data that could be easily utilized by third party services without worrying about how data are collected and stored and managed.

Keywords: Cloud computing, Internet of Things, FIWARE

1. Introduction

Over the years various services and applications have been developed in the concept of future Internet (FI) (9), (17). In particular such services are available by different cloud platform nodes such as FIWARE lab[1] that is a non-commercial sandbox environment. The services follow the form of RESTFul architecture (14) that allow to talk to each other easily and in a decoupled way. In addition, the Internet of Things (IoT) involves various sensors that are embedded to every day devices ans monitor data produced by humans or by the environment in an automatic way (20). The combination of cloud computing and IoT generates a new opportunity for wide discovery (18) and such data, since more and more FI applications are available. The development of such applications that are using cloud resources becomes more efficient (scalable storage) and create a significant impact on the economic benefits e.g. because of cloud elasticity and pay on demand model (11). In addition, the data transmission speed and the large volume of data (since cloud has the ability to store and process it) makes it even more attractive.

In this work we focus on the FI concept and especially on the FIWARE platform[2] that offers public services followed by simple application programming interfaces (APIs) to facilitate the process of developing smart applications. FIWARE motivates new entrepreneurs and software developers to implement such applications in health, environment and smart city concepts by providing Generic Enablers (GEs) (3) that are the building blocks of FI applications (20). In the general concept of a smart city many IoT devices

[1]https://www.fiware.org/lab/
[2]https://www.fiware.org

and sensors are associated with cloud computing services. For example in cases of data processing produced in order to avoid natural disasters (fires, floods, etc.), control of environmental conditions, energy saving, control of patient status and other. The sheer volume of data generated by sensors, has forced the transfer of the Internet of Things concept entirely on cloud computing since traditional systems could not handle the large volume of data as well as to guarantee remote access to other systems, e.g. for integration purposes.

The FIWARE lab[3] provides software to developers that use such services to develop smart applications/services within the smart city and thus including idea of the Internet of Things. Already in recent years, the community of FIWARE has taken important steps in the development of such services that help in creating more complex applications, however all these services are oriented with vendors, IoT devices and protocols. Having said that, this work proposes a Sensor Data Collection (SDC) cloud service that focuses on the problem of collecting data from different devices and their sensors, thus moving to a vendor agnostic solution. SDC developed as a gateway among IoT devices and cloud, enabling the collection of the different sensor signals that are eventually send to various other services. The service is designed to be extensible and generalized, so IoT devices could be easily connect and communicate without any programming intervention. Also, it is modular based on the service oriented architecture (2), that allows (a) support of multiple sensors belonging to different domains (for example medical, environmental etc.), and (b) support of network gateway devices.

The work is ordanized as follows. Section 2 presents the motivation and Sections 3 the related approaches to this study, Section 4 demonstrates the architecture of the SDC service, Section 5 presents an analysis of the implementation aspects and demonstration of the service API and Section 6 present the experimental analysis based on the simulation of two IoT devices that are (a) the Netatmo environmental sensor[4] and the Zephyr HxM Bluetooth Heart Rate Monitor medical sensor[5]. Finally, in Section 7 we conclude with the summary of this work and the future research directions.

[3]https://www.fiware.org

[4]https://www.netatmo.com/en-US/site

[5]http://www.zephyranywhere.com

2. Motivation

This work is based on FIWARE that is a non-commercial platform that offers general purpose services called Generic Enablers (GEs) that are in the form of APIs. In particular, GEs are provided by cloud computing infrastructure as SaaS (4) and if combined can constitute a special-purpose service called Specific Enablers (SEs), which could be used for developing solutions for more complex problem. FIWARE enables developers to obtain services as infrastructure (IaaS), creating virtual machines and allocating computing resources in the FIWARE lab (23).

FIWARE lab is based on the Openstack (5) platform that is an open source software, which allows the creation of a cloud computing systems. The latter are designed according to Openstack standards, thus consisting of a centralized architecture encompassing various smaller pieces of services that are responsible for controlling and managing the high volume computing resources (16). In this work we utilize and OpenStack system and FIWARE GEs to propose an architecture for a sensor data collection service on the cloud. The solution is modular, decentralized and reusable (19) thus allows IoT devices to easily to attach over the service. We are motivated by the works in publish/subscribe systems in clouds and inter-clouds as in (1), (8), (22), (7), (10) and (15).The basic characteristic of the proposed service is the simplicity of use at any time requested by the user. Such reusable services are very important in a cloud computing because it allows developers to model complex systems. Another important advantage is the modularity, that is to say the replacement of one individual service (GE) in case of a new version or a failure.

We implement our service within the IoT concept based on a service centric architecture as in (6) that is based on the fact that a large problem can be solved optimally and efficiently if it is divided into smaller parts. The advantages of such modular architectures are:

i. The services are reusable and can be made available on a larger scale.
ii. It provides faster and more efficient debugging and leading to improved fault tolerance.
iii. It involves shorter time with regards to the distribution of new products and applications.
iv. The services are not bounded to the system, thus can be easily replaced.
v. In case of integrating to a new system it does not require changes to the internal procedures of the service.

The SDC service has been developed in the form of the so called protocol adapters[6] that are implementations developed specifically for communication protocol (for example Wi-Fi, ZigBee, Bluetooth, etc.) as well as for specific devices. These services provide APIs, with functionalities such as data sending and alerting in case that a stimulus is generated from the systems. Also, it is possible to retrieve the characteristics of a device. Usually, the resulting response in a method that calls the service API is a standard data JSON (JavaScript Object Notation).

In this work we aim to develop a more generalized service that will allow data collection and storage from various IoT devices without worrying about protocols or device specifications. Thus, we aim to "transform" sensors to flexible APIs so data could easily be flown over the Internet to other services. Two main issues that the SDC service is focused are as follows.

i. The issue of having many different communication protocols between devices and network gateway. The main communication protocols on modern sensors are the Wi-Fi, Bluetooth, ZigBee, etc. Thus there is a need for a service that implements interfaces according to these standards, so to allow easily integration and communication between services and IoT devices.

ii. There is a huge variety of devices because companies provide proprietary APIs to collect data from the sensors, so the implementation of a service for commercial sensors seems quite tricky.

iii. The large volume of data produced by devices requires a new solution for scalable data storage. In addition, big data that are collected from different devices have different schemas thus a more sophisticated way is required for data storage.

The motivation of our work is based on the fact that to the best of our knowledge there is not a FIWARE service capable of managing, storing and sharing information in such way. Users who use this service may be persons, services and applications developed in FIWARE and other development environments. The proposed solution manages users and sensors for the immediate updating and subscribes users on data updates for each sensor. The basic functions supported are (a) add, remove and update sensors by the administrator, (b) add, remove and update user subscriptions, (c) add,

[6] http://catalogue.fiware.org/enablers/protocol-adapter-mr-coap

remove and update user rights in sensors assistance from the administrator, (d) update subscribers' sensors, (e) identification and delete users from the administrator and (f) database support with historical data belonging to different sensors.

Having said that, the contribution of this work includes the following.

i. The proposed architecture is dynamic and expandable, for example it could be easily integrated with services such as data analysis and processing.
ii. The service it self it could be used easily, is generalized to support multiple IoT devices and protocols and provides a flexible RESTFul API (14). This allows third party services and users to take advantage of the cloud technology and subscribe to IoT devices and their data remotely and on demand.
iii. The service is modular by separating front-end (IoT devices) and back-end (cloud system) and supports big data storage since it includes a scalable NoSQL database.
iv. It is compatible with any any service that supports a REST API e.g. with FIWARE services, thus it hides the internal service implementation details, its stateless and therefore easily scalable and provides loosely coupling.

Next Section 3 presents the works and technologies directly related with our study.

3. Related Works

As mentioned before, FIWARE platform provides software developers the necessary tools to build FI applications within the context of the smart city and of the IoT. FIWARE offers important benefits over the traditional systems including (a) elasticity as the platform could allow various levels of resource provisioning, (b) there is no need for software updates and maintenance, (c) increase accessibility and collaboration in terms of availability of services to 3rd party users and developers, (d) centralized security offered by the FIWARE platform (21) , (e) remote access from everywhere and anywhere through its powerful API that allows technology shift to seamless application development, and (f) customization and user tailored orientation through user personalized features (e.g., shared cloud storage collections for users) as described in (19) and (13).

In this work we utilize the following FIWARE GEs in order to integrate our solution:

i. Identity Management GE: This service covers certain aspects of users' access to networks, services and applications. Moreover, it is used for the authentication of third party service so to gain access to personal data stored in a secure environment. The KeyRock identity management[7] includes REST API interfaces for use by application developers. In our case the SDC service must register to the the KeyRock identity manager. It provides secure and private authentication from users to devices, networks and services, authorization and trust management, user profile management, privacy preserving disposition of personal data, Single Sign-On (SSO) to service domains and Identity Federation towards applications. Identity Management is used for authorizing foreign services to access personal data stored in a secure environment.

ii. JSON Storage GE: This GE supports information storage in JSON format through more abstract base type Mongo DB[8]. The service includes an API designed based on REST architecture. This GE provides NoSQL database management services through a REST API. Its users can perform CRUD (Create-Read-Update-Delete) operations on resources by using the basic HTTP methods (POST, GET, PUT, DELETE). In addition, it allows users to query over the stored resources.

iii. Publish/Subscribe Context Broker-Orion Context Broker GE[9]: The Orion Context Broker provides a publish/subscribe mechanism. Using the Orion Context Broker, users can subscribe to context elements (e.g., a room whose temperature and atmospheric pressure are measured) and get updated on context changes. In addition, they can use predefined conditions (e.g., an interval of time has passed or the context element?s attributes have changed) so they get context updates only when the condition is satisfied. This module allows users to subscribe to other users gesture collections to use them for building applications or to subscribe to a user who is broadcasting information.

   In this work we utilize the subscription request solution and we create

[7]http://catalogue.fiware.org/enablers/identity-management-keyrock

[8]https://www.mongodb.org

[9]http://catalogue.fiware.org/enablers/publishsubscribe-context-broker-orion-context-broker

a different entity for each sensor added to the system. Then users have the ability to make subscriptions to many features that are mapped to different sensors. In addition, by setting predetermined time interval or configure an on change paramater they can get data directly from the context broker. After a successful subscription, the user receives a response from the context broker with a unique subscription identification (id) since the system supports multiple requests from different users.

iv. Event Service Specific Enabler[10]: This SE is based on two entities namely as event senders and event receivers. The event senders are responsible for sending events in the SE and the event receivers (brokers) can make subscriptions to the SE. The Event Service model is demonstrated in Figure 1 and it supports publishing data in REST and XML-RPC format. This service is similar to the context broker and uses an API with graphical user interface support. However, security considerations have resulted in the extension of the service, as the current version does not guarantee the security of potentially sensitive data and how is managed and stored.

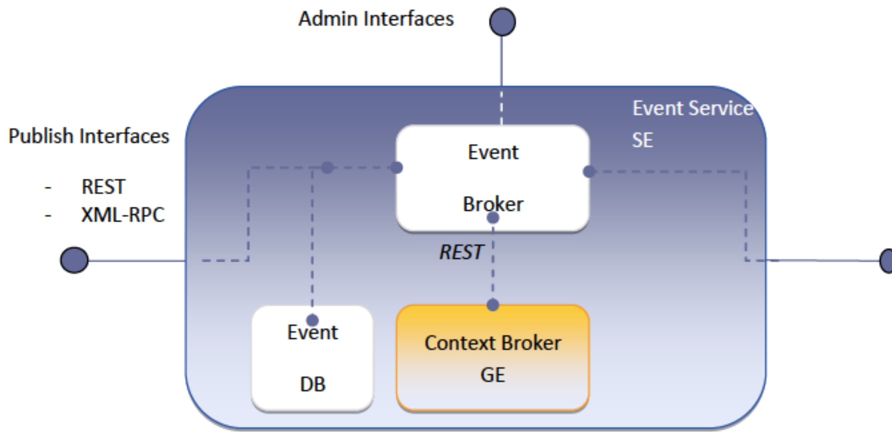

Figure 1: Event Service SE modular architecture

Also, it does not necessarily ensure data security due to non-use encryption algorithms (AES) for local storage. By contrast, during the conceptualization of the SDC model we use specific services offered by

[10]http://fistarcatalogue.fiware.eng.it/enablers/event-service

the FIWARE platform for the identification of users and data storage to ensure system security. By using remote storage e.g. utilizing the JSON Storage GE the SDC service made possible the creation of a public archive, where all instances can be accessed by the service. This was not possible in the Event Service GE because of the local storage. Therefore, the main differences of the SDC service with service Event Service GE are:

i. Identification of users who have an account in the FIWARE
ii. Remote Storage data service using a special storage service
iii. Ability licensing sensors, as not every user can be subscribed to specific sensors that determine the "administrator" of the "snapshot."

4. Architecting the Sensor Data Collection (SDC) Cloud Service

This section presents the reference architecture (Section 4.1) as a best practice model for IoT systems, the SDC architecture that derives from the reference model (Section 4.2) and the identification of the possible users (Section 4.3).

4.1. Reference Architecture

The reference architecture follows the service centric conceptualisation model as presented in (13). The reference architecture modules include producers (IoT devices), front-end (data collectors), back-end (resources for data storage and analysis) and consumers (third party users) and it is demonstrated in Figure 2.

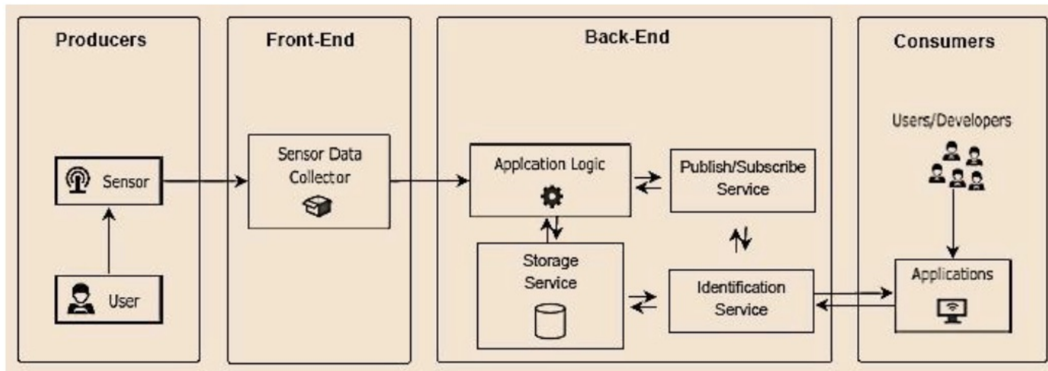

Figure 2: The reference architecture for sensor data collection in IoT systems

The architecture is divided into four parts that interact with each other in order to implement a generic IoT solution. The architecture parts are (a) the producers that include sensors that generate the information (e.g. data collection), (b) the front-end side, that plays the role of a gateway between the data sent by the sensor and the data managed by the application, (c) the back-end is the actual system and includes the general purpose services for user authentication, create subscription, data storage application and other (these integrate the application logic, which makes use of standards, controls and conditions for the transfer of information on individual services) and (d) the consumers are either end users or other applications that communicate with the SDC API. The architecture of the SDC service is derived from the reference model and is described bellow.

4.2. Architecture of the Sensor Data Collection (SDC) Cloud Service

This section presents the architecture of the SDC service that is based upon the following concepts.

i. Requires to maintain user data security based on validated user identification service.
ii. Provides scalable remote storage of user data, sensors and licenses by using a JSON storage service for secure and quick retrieval of data stored in the service.
iii. It is build upon the RESTFul architecture for efficient and flexible communication with other services. This will facilitate the development of more complex services and applications with extra functionality. Also it uses the JSON standard for information exchange.
iv. It exploits the benefits of cloud computing in the development of services such as elasticity, flexibility, and low infrastructure and maintenance costs.

To design the system we use the top-down approach, that is to begin with the overall system architecture and then analyze in detail the subsystems that compose it (2). The overall architecture is derived from the reference architecture and consists of different modules. The user interface (Front-End) connected directly to the system management interface (Back-End) and follow Users (Users) application following the conceptual model of Figure 2. Starting with the front-end this part is responsible for the connection of sensors and export of data to the cloud. As mentioned in the introduction, the communication protocol used by the sensors to communicate with the

SDC service varies from sensor to sensor, thus it is necessary to use a mechanism that will properly encode the data according to the standards set by the SDC service. This part includes the service functions implemented as insert/delete/update description of sensors and connect/disconnect of sensor. It further includes the method by which data are sent and related with administrator rights.

The back-end part is the system management interface that consists of general purpose services that we develop for the processing and storage of data and are the same for all sensors that interact with the system. More specifically, these are the Publish/Subscribe Orion Context Broker GE and JSON Storage GE (12), which is responsible for managing user subscriptions and storing information and data respectively. Furthermore, this part contains the mechanism for identification of users that enter the application, for example the KeyRock Identity Management GE. Finally the Application Logic orchestrates the transferring of the information to the appropriate individual parts (storage, identification and information manager). Figure 3 demonstrate the detailed architecture of the service. The four segments include user identification (authentication process), management information framework (context management), application logic and storage.

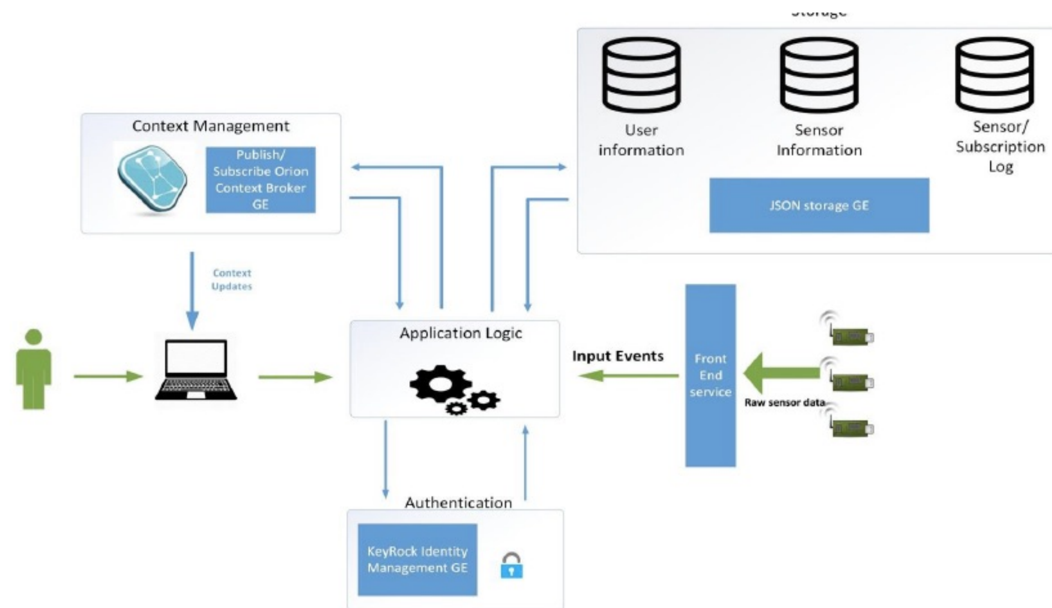

Figure 3: The sensor data collection service architecture

The modules of Figure 3 are described as follows.

i. The user identification (authentication) takes place on two levels, first through KeyRock Identity Management GE and then through the SDC service. In fact, the system administrator is responsible for identifying the consumer to access the functions of the SDC. This is due to the fact that the administrator should have full control of the snapshot system created.
ii. The context nanagement module ensures all the necessary processes such as creation of entity sensor (entity creation process), entity framework renewal (update context), create/update/delete user subscriptions and entity deletion/sensor. All these functions are included in RESTFul API offered by the publish subscribe Orion Context Broker GE.
iii. The application logic module includes the control, regulation and necessary API calls that the system is operating.
iv. The storage is used to store information about users (administrator and consumers), sensors and additional stored data history that have been sent from a sensor and user subscriptions history.

An important aspect of a generic IoT system is the orchestration of users with regards to their privileges for data accessibility. Next Section 4.3 details such conceptualization.

4.3. Users Orchestration

Here we identify the various user catagories that interact with the SDC and are either persons or IoT devices. Also we include services and applications that require to integrate functions of the SDC service in their own functionality offered via the API. In particular, users entering the system are twofold; the manager and the consumer. The first user who enters the system automatically receives administration rights, that is responsible for entering data into the system, as well as the management of sensors and users. The manager is therefore unique for each snapshot (instance) of the SDC service. On the other hand, the consumer is able to collect data from several sensors simultaneously or when sensors change some features (attributes) either on a threshold change or based on an interval. The user accessibility requirements are divided among managers and consumers and depending on the functionality that the developer want to map to each user.

Initially the manager is enrolled into the cloud platform and is the first user who logs into the system. In includes the following parameters.

i. It is unique for each snapshot of the SDC service and is responsible for the identification of users after the users have been identified by KeyRock IDM GE. If the user is not identified by the "manager" has no access to any function the API.
ii. It has the possibility of fixing permits the sensors to allow certain users to make contribution to specific sensors. Permission each sensor can include multiple users simultaneously and is unique.
iii. It has the possibility of accessing all sensors. It can also renew or delete the subscription. In case of renewal can only change the type of subscription and not the sensor data that wants to be informed.
iv. It is responsible for sending the data of the sensors in the system sensors management module.
v. It provides to connect or disconnect sensors in the system
vi. It recovers the schema of all connected sensors or a particular sensor.
vii. It creates a new sensor schema or deletes a particular sensor schema.
viii. It provides renewal of a particular sensor schema or recovers a specific sensor schema.
ix. It recovers all schemas of sensors and it provides recovering history of a particular schema.

The users management module includes the following parameters.

i. It provides recovery of all users of this "snapshot".
ii. It provides identification of a particular user to access the SE "consumer".
iii. It deletes a specific "consumer" of the SDC service.
iv. It provides recovery of all identified "consumer".
v. It provides retrieval of all unidentified "consumers".
vi. It authorizes a "consumer" with rights "administrator" and change rights of "manager" with rights "consumer".

The subscription management module includes the following parameters.

i. It retrieves all contributions made by all users.
ii. It recovers a specific subscription of a "consumer".
iii. It deletes a specific subscription of a "consumer".
iv. It creates a new own subscription.
v. It provides recovery mechanism.

vi. It deletes its own subscription.
vii. It provides renewal of its own subscription.

The license management module regards the permissions of users and it includes the following parameters.

i. It creates a license for a particular sensor.
ii. It recovers a license of a particular sensor.
iii. It deletes a license of a particular sensor.
iv. It permits renewal of a particular sensor.

The sensors data management module regards the permissions of users and it includes the following parameters.

i. Identifying the email address and password of the "manager", it is possible to send sensor data to the system.
ii. Recovery of data sensors depending on the contribution it has made.

The consumers are also enrolled into the cloud platform and their role includes the following parameters.

i. To access any of the following functions to be initially identified by KeyRock Identity Mangement GE with the email address and password, and secondly from the "manager" of the "snapshot (instance).
ii. To make a sybscription to the sensor data will first need to have the permission of the "manager"
iii. Has the possibility of subscription to all sensors depending on the manager aproval. It can also renew or delete the subscription. In case of renewal can only change the type of assistance and not the sensor data that wants to be informed.
iv. It is the user who has the right to subscribe sensors, for which he has been given permission from the "manager". There is the possibility of sending data of the sensors in the system.

The consumers sensor management module includes the following parameters.

i. Recovery of shapes all connected sensors
ii. Recover the shape of a particular associated sensor
iii. Recover a specific sensor shape

iv. Recover all shapes of sensors
v. Recovering history of a particular sensor

The consumers subscription management module includes the following parameters.

i. Create a new own assistance
ii. Recovery of own contribution
iii. Delete its own assistance
iv. Renewal of its own assistance License Management (Permissions Management)
v. Recovery licenses for a particular sensor

The consumers data management module includes the following parameters.

i. Recovery of data sensors depending on the subscriptions it has made.

Next Section 5 details the implementation aspects and configurations of the SDC service.

5. Implementation of the Sensor Data Collection Cloud Service

This section presents the implementation of the solution and the RESTFul API. We detail the methods and the functionality that each of which performs. To characterize the API, we classify its functionalities into five categories similar to the requirements analysis phase. These categories of the API methods are (a) the management of sensors, (b) the management of users, (c) the management of subscriptions,(d) the management of licenses and (e) the management of sensor data. Finally, we present a discussion of the implementation of the user authentication functionality.

5.1. Management of sensors

This section presents the functionalities of the module related with the management of devices (and their sensors) e.g. on how to create, read, update and delete sensors. These are presented in the form of the method (e.g. GET /sensors) and the explanation of its functionality.

- GET /sensors

(1) It check if the user is identified by the manager, the HTTP call is GET `http://URL:3000/users/public_user/dbs/PredefinedSensors/collections/predefinedSensors/records` to the JSON Storage GE for retrieving the available schemes of the sensors. In case of a call failure, the system replies back with a *CurlException* type error.
(2) The method returns the result of the call to the user.

- GET /sensors/sensorId

(1) It checks if the user is identified by the manager.
(2) It check if there is a sensor with unique identifier sensorId at `http://URL:3000/users/public_user/dbs/PredefinedSensors/collections/predefinedSensors/finds`. If there is a result this returns to the user, otherwise the system responds with *NotFoundException* type error.

- POST /sensors

(1) It checks if the user is the administrator.
(2) It checks if the data received by the SDC via POST, are in JSON format and additionally in the form determined by the user. The data in this method should be of the form: "name": "Atmo", "attributes": ["name": "temperature", "type": "celsius", "name": "pressure", "type": "bar"] In this example *Atmo* is the sensor name followed by the characteristics of which the user takes measurements. Also the method checks if the names of the features are unique to avoid a mistake on possible errors of such a feature. If the data are not in JSON format system responds with *JsonException* type error and if they are not in the above format or the names of features are not unique it replies with *DataFormatException* type error.
(3) The system automatically given a unique identifier (id) for this sensor, so that we can have multiple sensors of the same type for example for sensor named as *Atmo* will assign a different identifier e.g. Atmo1.
(4) After hte call is made at `http://URL:3000/users/public_user/dbs/PredefinedSensors/collections/predefinedSensors`

`/records`. The POST method is in the JSON Storage GE and the sent (e.g. to insert a new sensor) are stored in the collection *predefined sensors* that is a collection available to all users so to avoid duplicate entries of the same sensors. In case of any error in the method call the JSON Storage GE system responds with *CurlException* type error. In case of a successful POST call, the system responds with message "response": "The sensor added successfully".

- PUT /sensors/sensorId
  (1) It checks if the user is the system administrator.
  (2) It check if the sensor with unique ID (*sensorId*), is in the collection *predefinedSensors*. If not, the system replies with *NotFoundException* type error.
  (3) If the sensor ID sensorId is connected, the system responds with *NotAllowedException* type error. If it is not connected it checks the data received from the user. Also, if it is in the form of JSON or it is not in the form determined by the user (according to the sensor schema) the system responds with a*JsonException* or *DataFormatException* respectively. The user data must be in the following JSON form: "name": "Atmo", "attributes": ["name": "temperature", "type": "celsius", "name": "pressure", "type": "bar"]
  (4) If the data is in accordance with the specifications then the service refreshes the schema of the sensor to collect predefinedSensors of JSON storage GE and the system responds with "response":"The sensor updated successfully".

- DELETE /sensors/sensorId
  (1) It checks if the user is the administrator.
  (2) It checks if the sensor with unique ID (*sensorId*), is at the collection *predefinedSensors*. If not then the system replies with *NotFoundException* type error.
  (3) If the sensor ID (*sensorId*) is validated correctly, then the system deletes the connected sensor from the collection *connectedSensors* and from the collection *predefinedSensors*. If deleted without error the system "reponds "response": "Sensor has been deleted successfully", otherwise it displays a *CurlException* type error.

- GET /connected/sensorId
  1. (1) It checks if the user is identified by the manager.
  2. (2) It checks if the sensor with unique ID (*sensorId*), is at the collection *connectedSensors*. If not the system replies with *NotFoundException* type error. If there a result, the particular sensor is displayed to the user.

- GET /connected
  1. (1) It checks if the user is identified by the manager.
  2. (2) It makes a call the the JSON storage and its *connectedSensors* collection and the result of the call is displayed to the user. If there is an error the system responds with *CurlException* type error.

- DELETE /connected/sensorId
  1. (1) It checks if the user is manager.
  2. (2) It checks if the sensor with unique ID (*sensorId*), belongs at the collecting *connectedSensors*. If not the system replies with *NotFoundException* type error. If there a result, the particular sensor is displayed to the user.
  3. (3) After the call is directed to the JSON Storage GE for deletion from the collection connectedSensors. In case of an error, the system responds with a *CurlException* type error.
  4. (4) Then the method deletes the entity of this sensor from the Context Broker GE by making a call at `http://URL:1026/NGSI10/contextEntities/{id}` with id the sensor that the user wants to delete. If the deletion was successful then the system responds with "response": "The connected sensor deleted successfully". In case of an error in the call to the Context Broker GE system it responds with *CurlException* type error.

- /sensors/sensorId/log
  1. (1) It checks if the user is the manager.
  2. (2) It checks if the sensor with unique ID (*sensorId*), belongs at the collecting *connectedSensors*. If not the system replies with *NotFoundException* type error. If there a result, the particular sensor is displayed to the user.

(3) Then the method makes a call is in JSON Storage GE based on the log collection and the sensor id so the data is displayed to the user. In case of an error in the call it responds with *CurlException* type error.

5.2. Management of users

This section presents the functionalities of the module related with the management of users e.g. on how to create, read, update and delete users. These are presented in the form of the method (e.g. GET /users) and the explanation of its functionality.

- GET /users
  (1) It checks if the user is the manager.
  (2) It makes a call in the JSON Storage GE and in the collection of users *Base Users*, the method displays all the users of that snapshot. In case of an error in the call it responds with *CurlException* type error.

- GET /user/userId
  i. It checks if the user is the manager.
  ii. It checks if the user with unique ID *userId*, is in the collection *users*. In case of an error in the call it responds with *NotFoundException* type error.

- POST /user/admin
  (1) It checks if the user is the manager.
  (2) It checks if the data sent by the user by calling the POST method in the form of JSON and according to the system specifications. User data must be of the form: "email": "123@gmail.com"
  (3) It checks if the email that gives the user exists in the collection of users via a call on JSON Storage GE. If there is no user by email the system responds with *NotFoundException* type error. If there is an error in the validation of the user from the administrator, the system responds with *InvalidTypeException* type error. In case of a successful call to get users from JSON Storage GE, the system change the rights for an administrator to consumer based on the given email and it displays back to the user the message "response": "The change of administrator completed successfully".

- GET /users/unauthorized
  (1) It checks if the user is the manager.
  (2) It makes a call to the JSON Storage GE collection *Base Users*, and the system displays all users that are authorized. In case of an error in the call it responds with *CurlException* type error.

- GET /users /authorized
  (1) It checks if the user is the manager.
  (2) It makes a call to the JSON Storage GE collection *Base Users*,and the system displays all users that are authorized. In case of an error in the call it responds with *CurlException* type error.

- POST /user /authorize
  (1) It checks if the user is the manager.
  (2) Check if the data sent by the user by calling the POST method in the form of JSON and according to the system specifications. User data must be of the form: "email": "123@gmail.com"
  (3) Check if the email that gives the user exists in the collection of users via a call on JSON Storage GE. If there is no user by email the system responds with *NotFoundException* type error. If you find a user with this email then it checks if it is the manager or is identified by the manager as a consumer. If it is not one of both, the system responds with *NotAllowedException* type error. Then, in a successful call to collect users from the JSON Storage GE, it identifies the consumer with the email and displays back to the user the message "response": "User has been authorized".

- DELETE /user/userId
  (1) It checks if the user is the manager.
  (2) It checks if the user with unique ID *userId* in the collection *users*. If not, then the system replies with *NotFoundException* type error. If there is then it checks if the user is consumer or administrator. If is manager, it appears *NotAllowedException* type error if is consumer then with two calls to JSON Storage GE deleted the user from the collection of users, and then deletes the historical data of the subscriptions the user has from the collection *SubscriptionLog*. Finally, the user is shown the message "response": "The user has been successfully deleted".

5.3. Subscription Management

This section presents the functionalities of the module related with the management of subscriptions e.g. on how to create, read, update and delete subscriptions. These are presented in the form of the method (e.g. GET /subscriptions) and the explanation of its functionality.

- GET /subscriptions
  (1) It checks if the user is the manager.
  (2) It makes a call to the JSON Storage GE to collect subscriptions from collection *Users*, adn the method displays the user subscriptions. In case of an error in the call it responds with *CurlException* type error.

- GET /subscriptions/subscriptionId
  (1) It checks if the user is the manager.
  (2) It checks if there is a record with unique ID *subscriptionId* at the collection *subscriptions*of JSON Storage GE. If not then the system replies with *NotFoundException* type error. If there is then it responds back to the user. In case of an error in the call it responds with *CurlException* type error.

- GET /subscriptions/log
  (1) It checks if the user is the manager.
  (2) Using the function *getUserInfo()*, the method gets the data of the user for its unique ID *userId*. Then there is a call in JSON Storage GE collection to identify the *userId* in the collection *SubscriptionLog*. Finally, the method responds with the history of data for sensor usage that the user has been subscribed in the past.

- DELETE /subscriptions/subscriptionId
  (1) It checks if the user is the manager.
  (2) It checks if the record with unique ID *subscriptionId* is at the collection *subscriptions* of the JSON Storage GE. If not then the system replies with *NotFoundException* type error.

(3) It calls the JSON Storage GE to delete the subscription from the collection *subscriptions*. Then makes a call at `http://URL:1026/NGSI10/unsubscribeContext` that is the Context Broker GE to delete the user's subscription.

- GET /subscription

  (1) It checks if the user is the manager.
  (2) Initially it gets data for the identified user using the cache and it checks if the user already is on the subscription system. If not then the system replies with *NotFoundException* type error. But if the user is subscribed then it call the JSON Storage GE and the collection *subscriptions* to retrieve and display the subscription data of the specific user. If an error occurs during the call to the JSON Storage GE system responds with error type *Curl Exception*.

- POST /subscription

  (1) It checks if the user is the manager.
  (2) It checks if the data sent to the user by calling the POST method in the form of JSON and according to the system specifications. The user's data is twofold, depending on the type of the required subscription. If the user wants to receive sensor data when changing the value of a characteristic of the sensor, the characteristic (property) *subType* data sent by the user must be *ONCHANGE*. The attribute, belonging to the sensor that we want to change is specified as *condAttributes*. In *condAttributes* the user can enter more than one characteristics and when one of them is changed, the user is informed. For *subType* = *ONCHANGE*, the form of user data must be: "subAttributes": ["name": "a1", "sensorid": "test1", "name": "a2", "sensorid": "test1"], "subType": "ONCHANGE", "condAttributes": ["name": "a1", "sensorid": "test1"] However, if the user wishes to be updated on a time interval the characteristic (property) *subType* requires to have the property *ONTIMEINTERVAL*. In this case it is required to define another attribute (property) interval. Interval is the time in seconds after the conclusion of which the user receives updates from the Context Broker GE for sensor data that has subscribed. For

subType = ONTIMEINTERVAL, the form of user data must be: "subAttributes": ["name": "a1", "sensorid": "test1", "name": "a2", "sensorid": "test1"], "subType": "ONTIMEINTERVAL", "interval": "3"

(3) Then, making sure that the data you provide us with the user in accordance with the standards that tje service has set, a check regarding the special permits established by the manager. If for example a sensor that the user is subscribes has not been determined to have subscription license to the user then the system responds with *SubscribeException* type error. If could perform subscription then execution continues at the next step.

(4) In the next step the user data is appropriately configured to make the call to the Context Broker GE for requesting subscription to the `http:/URL:1026/NGSI10/subscribeContext`.

(5) After of a successfully performed call to the Context Broker, the method call the JSON Storage GE to collect subscriptions.

- DELETE /subscription

  (1) It checks if the user is the manager.

  (2) It uses the function *getUserInfo()*, the method gets the user's unique identifier subscription *subId*. If the user is not subscribed (*subId = NULL*) the system responds with *NotFoundException* type error, while if there is then execution continues to the next step.

  (3) It makes a call to the JSON Storage GE to delete the subscription from the collection named as *subscriptions*. Then call is at `http://URL:1026/NGSI10/unsubscribeContext` the Context Broker GE to delete the user's subscription and it does not sent more data updates for subscription.

- PUT /subscription

  (1) It checks if the user is the manager.

  (2) It uses the function *getUserInfo()*, the method gets the items as the user's unique identifier assistance *subId*.If the user is not subscribed (*subId = NULL*) the system responds with *NotFoundException* type error, while if there is then execution continues to the next step.

(3) It checks if the data sent to the user by calling the POST method in the form of JSON and according to the system specifications. The user's data is twofold, depending on the type of the required subscription. If the user wants to receive sensor data when changing the value of a characteristic of the sensor, the characteristic (property) *subType* data sent by the user must be *ONCHANGE*. The attribute, belonging to the sensor that we want to change is specified as *condAttributes*. In *condAttributes* the user can enter more than one characteristics and when one of them is changed, the user is informed. For *subType = ONCHANGE*, the form of user data must be: "subType": "ONCHANGE", "condAttributes": ["name": "a1", "sensorid": "test1"] However, if the user wishes to be updated on a time interval the characteristic (property) *subType* requires to have the property *ONTIMEINTERVAL*. In this case it is required to define another attribute (property) interval. Interval is the time in seconds after the conclusion of which the user receives updates from the Context Broker GE for sensor data that has subscribed. For *subType = ONTIMEINTERVAL*, the form of user data must be: "subType": "ONTIMEINTERVAL", "interval": "3" The sensor data which subscribed can not be renewed by calling that method since the Context Broker GE does not offer this feature. The method can only update the type of subscription.

(4) After the user data is appropriately configured to make the call to the Context Broker GE for the renewal of an existing subscription at `http://URL:1026/NGSI10/subscribeContext`.

(5) After successfully performed the call to Context Broker, then call is in JSON Storage GE to collect subscriptions, to renew the existing subscription.

5.4. Management of licenses

This section presents the functionalities of the module related with the management of licenses e.g. on how to create, read, update and delete licenses. These are presented in the form of the method (e.g. GET /sensors/sensorId/permissions) and the explanation of its functionality.

- POST /sensors/sensorId/permissions

  (1) It checks if the user is the manager.

(2) It checks if the sensor with the unique ID *sensorId* exists in the collection *predefinedSensors* in the JSON Storage GE. If not the method returns a *NotFoundException* type error. Then it check if the particular sensor has been used by some users. If a license is not stored it displays a user *PermissionException* since the license of each sensor is an entry in the JSON Storage GE. Otherwise, execution continues to the next step.

(3) It controls data given by the user through ta validation process, to be in JSON format and conform to specifications. User data must be of the form: "users": ["123@gmail.com", "kdmortal@gmail.com"] In characteristic (property) of JSON users are given all emails of the users allows the mamager to make subscription for sensor data with ID *sensorId*.

(4) It checks if the emails are mapped to the users who are actually application users. If not the system responds with a *NotFoundException* type error.

(5) The final step of this process is to store the license of this sensor on the collection *Permissions* and collects the *sensorId* from the JSON Storage GE.

- GET /sensors/sensorId/permissions

  (1) It checks if the user is the manager.

  (2) It checks if the sensor with unique ID *sensorId* exists in the collection predefinedSensors JSON Storage GE. If there not there is a *NotFoundException* type error. Then it checks if the particular sensor has been used by some users. If there is it displays back to the user, all the users who can make contribution in this sensor.

- DELETE /sensors/sensorId/permissions

  (1) It checks if the user is the manager.

  (2) It checks if the sensor with unique ID *sensorId* exists in the collection *predefinedSensors* in the JSON Storage GE. If there is not it responds with a *NotFoundException* type error. Then it checks if the particular sensor has been used by some users. If there is, then makes a call to the JSON Storage GE to delete the collection for sensorId *sensor* .

- PUT /sensors/sensorId/permissions
  (1) It checks if the user is the manager.
  (2) It checks if the sensor with unique ID *sensorId* exists in the collection *predefinedSensors* in the JSON Storage GE. If there is not it responds with a *NotFoundException* type error. Then check if the particular sensor has been used by some users. If there is no stored license then it responds with a *PermissionException*. Otherwise, execution continues to the next step.
  (3) The method validates the data given by the user through the process, to be in JSON format and conform to specifications. User data must be of the form: "users": ["123@gmail.com", "kdmortal@gmail.com"] In characteristic (property) of JSON users are given all emails of the users and the manager makes subscription for sensor data with ID sensorId.
  (4) It checks if emails are users who are actually given access to applications. If not the system responds with *NotFoundException* type error.
  (5) Then renew the already existing authorization of this sensor on the collection *Permissions* and collecting the *sensorId* from the JSON Storage GE.

5.5. Management of sensors data

This section presents the functionalities of the module related with the management of sensors data e.g. on how to create, read, update and delete sensors data. These are presented in the form of the method (e.g. GET /contextNotifications) and the explanation of its functionality.

- GET /contextNotifications
  (1) Using this method the user is informed in real time for sensor data to which it has been subscribed. For the operation of the function we use the *notifyContext()* which is connected to the updates of the Context Broker GE. To implement the *notifyContext* we used server side events that asynchronously sends data to the user when on request from the Context Broker GE. The operation of this method is demonstrated in figure 4.
  (2) It checks if the user is identified by the manager.

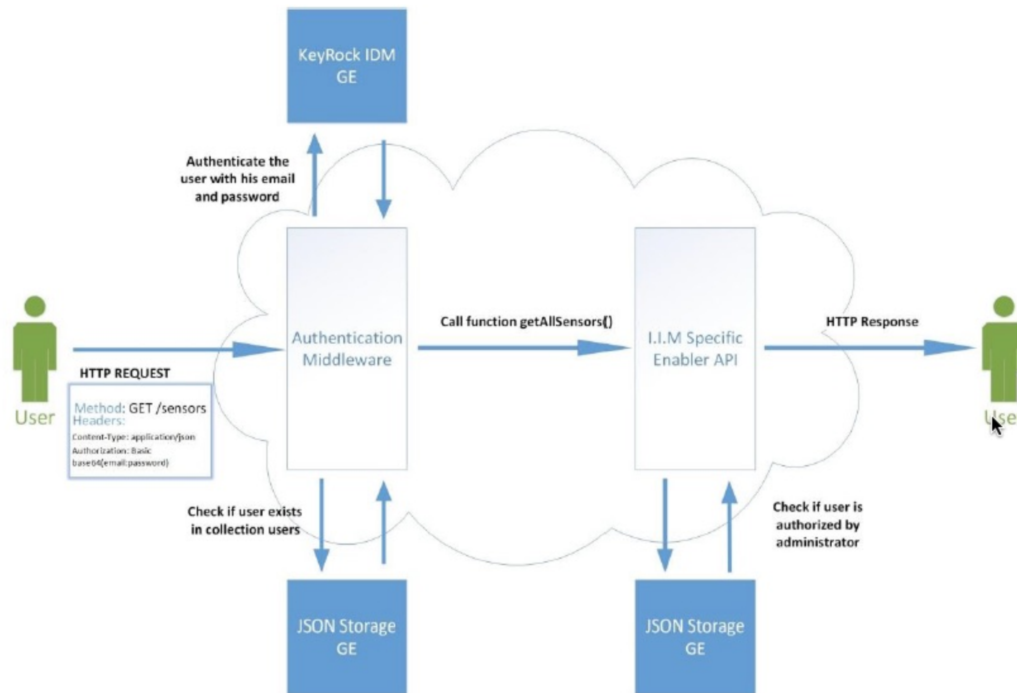

Figure 4: Context notifications on real time

(3) The method checks if the user has already a subscription in the SDC service, in any other case the system displays a *NotFoundException* type error.

(4) The system replies to the user based on the *notifyContext()*, based on the reception of updates from the Context Broker GE (these are sensor data that are collected in real time). The data displayed in response of the user is in the form of JSON.

- POST /event

  By using this method the manager has the ability to send sensor data to the system. It includes two functions the *checkEvent*, which controls whether the data sent by the manager is JSON, are in accordance with the specifications and additional line if the sensor shape. The second function is the *dataReciever*, which makes the necessary calls to Context Broker to create or update the entity and the JSON Storage GE sensor for storing collection connectedSensors, storing sensor data,

etc. Detailed implementation of the method described in the following steps:

(1) It checks if the user is a manager.
(2) It controls the data sent by the user through the function *checkEvent*, to be in JSON format, according to the sensor format and according to the standards we set. User data must be of the form: "id": "Atmo", "attributes":["name":"temperature","type": "celsius","value":"20",
"name":"humidity","type":"percentage","value":"75"] The id is the unique sensor ID and the characteristic attributes include sensor data with the latest prices.
(3) Then a check is made whether the *sensorid* is given by the user is stored in the collection of *predefinedSensors* of the JSON Storage GE. If you do not find shape sensor with id it is sent by the user, it shows a *NotFoundException* type error. Otherwise, execution continues to the next step.
(4) Then it checks if the sensor is already connected. If not then there is a call in the Context Broker to create the entity. Otherwise simply renew the stored entity by performing a call to Context Broker at `http://URL:1026/NGSI10/updateContext`.

5.6. Implementation of the user authentication functionality

This sections presents the user authentication middleware that is offered by the SDC service. The middleware is called *HttpBasicAuth* and allows us to identify the user at every call of a method of API. Whenever the user makes call to a method of the API, it requires to provide credentials (in the form of username and password). The username and password are coded in the form of base64 (username: password) and be placed as a header to the HTTP request made by the user. We implement the SDC to utilize two middleware, (a) the first is used for identification and the second for caching. In our own service using a Middleware for identifying users. Figure 5 demonstrates the authentication middleware for user identification.

The middleware it self consists of functions to authenticate and store the user. Originally the function specifies that the requested API address URL/api/ does not required user authentication. After the function is called to authenticate through the appropriate method of the KeyRock Identity Management GE `https://account.lab.fiware.org/api/v1/tokens.json?`

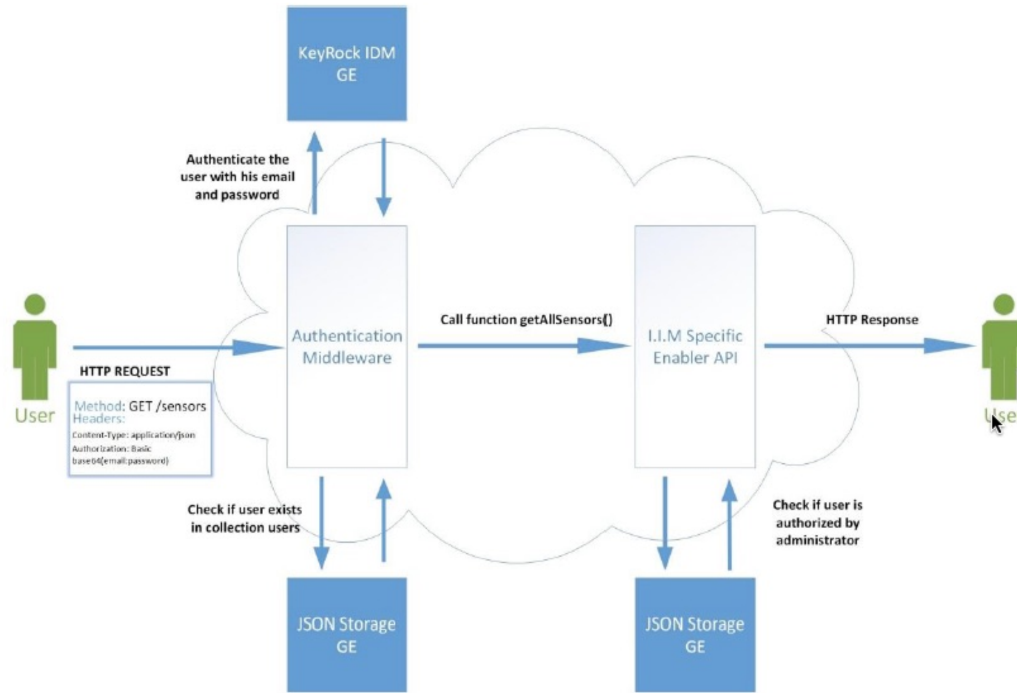

Figure 5: Utilization of authentication middleware for user identification

`email=123@gmail&password=123` using the email and password. According to the API of the KeyRock call this method has resulted in the identification of the user and gives a response to the user with a session token and a status code 200 OK.

Once the user is identified it checks if it has already been stored in the *Users* collection. If it is already stored then ends the execution of middleware and the implementation of the program shifted to the execution of API. If the user is not stored in the users collection then first a check is made to determine whether it has rights as manager or not and then stored in the collection. At that point ir ends the execution of the middleware and called the function for the execution of the main API.

6. Experimental Analysis

This section presents the experimental analysis of the work in order to demonstrate the effectiveness of the proposed solution. We utilize two IoT de-

vices that are (a) the Netatmo (Environmental sensor)[11] and (b) the Zephyr HxM Bluetooth Heart Rate Monitor (medical sensor)[12]. The Netatmo sensor is monitoring climatic changes (temperature, humidity, etc. pressure). The device supports services for storage and processing of data to a private cloud infrastructure. The sensor is met in applications for remote environmental control and anticipation of natural disasters such as fire or frost. The Zephyr HxM Bluetooth Heart Rate Monitor uses the Bluetooth communication protocol, and sends to a mobile device medical data of the person who wears it. It can take measurements such as heart rate, speed, distance and time between two peaks in an electrocardiogram.

We test the performance of the SDC iby simulating the two IoT devices and we check whether can the SDC to respond fully functional on a continuous sensor data. The experiment we conducted is performed in four phases. Each phases includes two users who subscribe to data provided by two sensors in the system, the Zephyr is medical sensor and measures the heart rate in bpm (beats per minute) and environmental Atmo sensor, which measures the temperature in centigrades and humidity in percentage. To simulate the operation of these sensors we created a simulation software for each of which and we send sensor data to a 5 minutes window at a time set by each phase of the experiment.

The time data refresh intervals of the sensors cover a wide range of 15 seconds in the first phase until 2 seconds in 4th phase. Table 1 demonstrates the experimental analysis and results.

Table 1: Experimental results for simulations of sensors Zephyr and Atmo

Device	Zephyr	Atmo	Results Zephyr	Results Atmo
1st phase	15	15	1.6	1.6
2nd phase	10	10	1.6	1.6
3rd phase	5	5	1.7	1.7
4th phase	Random (2 to 4)	Random (2 to 4)	1.8	1.8

Notably, the lower limit to determine the time was 2 seconds as a shorter period would mean losing the data sent to the SDC (by the execution of the

[11]https://www.netatmo.com/en-US/site

[12]http://www.zephyranywhere.com

method POST /event with an average response time of 1.6 to 1.8 seconds). Below we present the results of the experiment, which is the time difference between the time that the call is made in the SDC (call POST /event) for sending sensor data and timing data that is displayed to the user (GET /contextNotifications).

We conclude that in a small refresh interval data, and if we do not greatly affected the difference in times between the sending data and display to the user, we notice a little difference because of the burden of the continuous data. This time difference is between 1.6 to 1.8 seconds. This is because the function is executed POST /event to the API. Also is because of the checks done in JSON for the user and because the calls to the various modules has a fixed time delay of 1.6 to 1.8 seconds. This delay is justified by the requests made, in the Context Broker GE with an average response time of 700 ms and the two requests made in the JSON Storage GE with an average response time of 500 ms. Finally, if we include in these the delays due to the network it is apparently that the time delay of 1.6 to 1.8 seconds could be considered as realistic.

7. Conclusions

The SDC service developed to allow efficient, fast and on the fly IoT data collection and storage to a cloud system. Our solution allows the efficient management of users and sensors and an on the fly updating of the subscribers (users) with regards to data updates of each sensor. We take advantage of the benefits offered from the combination of these technologies. The next list demonstrates the basic functionality of the services.

1. The service allows to add/remove/update sensors by the administrator.
2. It allows to add/remove/update user subscriptions.
3. Administrator can add/remove/update user rights for sensors.
4. It includes a database with historical data for the sensors. Since the database is scalable (e.g. MongoDB) it could easily include big data.
5. The service is expandable and can be added to other services that require to expand their functionality such as data analysis tools.
6. The architecture is modular and the solution is easy to use, based on the RESTFul API that is available for utilization over the Internet.
7. The solution is compatible with FIWARE thus could be easily integrated to other cloud applications.

An important aspect of the system was the use of FIWARE GEs thus we were able to develop a modular solution that meets the specifications of a modern FI application. The future research directions include the definition of patterns to specific sensor data thus the users could be notified according to patterns and rules. For example, if the temperature of an internal space continuously rises over 5 minutes then the user is alerted. Another aspect is to ensure further security in the data management by the service. Finally, we aim to explore scaling behaviour of the JSON storage module in order to experiment how big data affects performance of the system.

8. References

[1] A. Antonic, M. Marjanovic, K. Pripui, and I. P. arko. A mobile crowd sensing ecosystem enabled by cupus: Cloud-based publish/subscribe middleware for the internet of things. *Future Generation Computer Systems*, 56:607 – 622, 2016.

[2] J. Bih. Service oriented architecture (soa) a new paradigm to implement dynamic e-business solutions. *Ubiquity*, 2006(August):4:1–4:1, Aug. 2006.

[3] J. Brogan and C. Thuemmler. Specification for generic enablers as software. In *Information Technology: New Generations (ITNG), 2014 11th International Conference on*, pages 129–136, April 2014.

[4] V. Chang. The business intelligence as a service in the cloud. *Future Generation Computer Systems*, 37(0):512 – 534, 2014. Special Section: Innovative Methods and Algorithms for Advanced Data-Intensive Computing Special Section: Semantics, Intelligent processing and services for big data Special Section: Advances in Data-Intensive Modelling and Simulation Special Section: Hybrid Intelligence for Growing Internet and its Applications.

[5] A. Corradi, M. Fanelli, and L. Foschini. {VM} consolidation: A real case based on openstack cloud. *Future Generation Computer Systems*, 32(0):118 – 127, 2014. Special Section: The Management of Cloud Systems, Special Section: Cyber-Physical Society and Special Section: Special Issue on Exploiting Semantic Technologies with Particularization on Linked Data over Grid and Cloud Architectures.

[6] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[7] C. Esposito, M. Ficco, F. Palmieri, and A. Castiglione. Interconnecting federated clouds by using publish-subscribe service. *Cluster Computing*, 16(4):887–903, 2013.

[8] C. Esposito, M. Ficco, F. Palmieri, and A. Castiglione. A knowledge-based platform for big data analytics based on publish/subscribe services and stream processing. *Knowledge-Based Systems*, 79:3 – 17, 2015.

[9] A. Galis and A. Gavras. *The Future Internet: Future Internet Assembly 2013 Validated Results and New Horizons*. Springer Publishing Company, Incorporated, 2013.

[10] X. Ma, Y. Wang, Q. Qiu, W. Sun, and X. Pei. Scalable and elastic event matching for attribute-based publish/subscribe systems. *Future Generation Computer Systems*, 36:102 – 119, 2014. Special Section: Intelligent Big Data ProcessingSpecial Section: Behavior Data Security Issues in Network Information PropagationSpecial Section: Energy-efficiency in Large Distributed Computing ArchitecturesSpecial Section: eScience Infrastructure and Applications.

[11] D. Petcu. Consuming resources and services from multiple clouds. *J. Grid Comput.*, 12(2):321–345, June 2014.

[12] A. Preventis, K. Stravoskoufos, S. Sotiriadis, and E. G. M. Petrakis. Interact: Gesture recognition in the cloud. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, UCC '14, pages 501–502, Washington, DC, USA, 2014. IEEE Computer Society.

[13] A. Preventis, K. Stravoskoufos, S. Sotiriadis, and E. G. M. Petrakis. Personalized motion sensor driven gesture recognition in the fiware cloud platform. In *Proceedings of the 2015 14th International Symposium on Parallel and Distributed Computing*, ISPDC '15, pages 19–26, Washington, DC, USA, 2015. IEEE Computer Society.

[14] S. Schreier. Modeling restful applications. In *Proceedings of the Second International Workshop on RESTful Design*, WS-REST '11, pages 15–21, New York, NY, USA, 2011. ACM.

[15] A. Sfrent and F. Pop. Asymptotic scheduling for many task computing in big data platforms. *Information Sciences*, 319:71 – 91, 2015. Energy Efficient Data, Services and Memory Management in Big Data Information Systems.

[16] S. Sotiriadis and N. Bessis. An inter-cloud bridge system for heterogeneous cloud platforms. *Future Gener. Comput. Syst.*, 54(C):180–194, Jan. 2016.

[17] S. Sotiriadis, N. Bessis, and N. Antonopoulos. Towards inter-cloud schedulers: A survey of meta-scheduling approaches. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2011 International Conference on*, pages 59–66, Oct 2011.

[18] S. Sotiriadis, N. Bessis, Y. Huang, P. Sant, and C. Maple. Towards decentralized grid agent models for continuous resource discovery of interoperable grid virtual organisations. In *Digital Information Management (ICDIM), 2010 Fifth International Conference on*, pages 530–535, July 2010.

[19] S. Sotiriadis, N. Bessis, and E. Petrakis. An inter-cloud architecture for future internet infrastructures. In F. Pop and M. Potop-Butucaru, editors, *Adaptive Resource Management and Scheduling for Cloud Computing*, Lecture Notes in Computer Science, pages 206–216. Springer International Publishing, 2014.

[20] S. Sotiriadis, E. Petrakis, S. Covaci, P. Zampognaro, E. Georga, and C. Thuemmler. An architecture for designing future internet (fi) applications in sensitive domains: Expressing the software to data paradigm by utilizing hybrid cloud technology. In *Bioinformatics and Bioengineering (BIBE), 2013 IEEE 13th International Conference on*, pages 1–6, Nov 2013.

[21] A. G. Vázquez, P. Soria-Rodriguez, P. Bisson, D. Gidoin, S. Trabelsi, and G. Serme. Fi-ware security: Future internet security core. In *Proceedings of the 4th European Conference on Towards a Service-based Internet*, ServiceWave'11, pages 144–152, Berlin, Heidelberg, 2011. Springer-Verlag.

[22] X. Xie, H. Wang, H. Jin, F. Zhao, X. Ke, and L. T. Yang. Dta: Dynamic topology algorithms in content-based publish/subscribe. *Future Generation Computer Systems*, 54:159 – 167, 2016.

[23] T. Zahariadis, A. Papadakis, F. Alvarez, J. Gonzalez, F. Lopez, F. Facca, and Y. Al-Hazmi. Fiware lab: Managing resources and services in a cloud federation supporting future internet applications. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, UCC '14, pages 792–799, Washington, DC, USA, 2014. IEEE Computer Society.