Proceedings of the 51st Hawaii International Conference on System Sciences | 2018

Evaluating the FIWARE Platform

A Case-Study on Implementing Smart Application with FIWARE

Peter Salhofer
FH JOANNEUM
peter.salhofer@fh-joanneum.at

Abstract

This paper describes the result of a thorough analysis and evaluation of the so-called FIWARE platform from a smart application development point of view. FIWARE is the result of a series of well-funded EU projects that is currently intensively promoted throughout public agencies in Europe and world-wide. The goal was to figure out how services provided by FIWARE facilitate the development of smart applications. It was conducted first by an analysis of the central components that make up the service stack, followed by the implementation of a pilot project that aimed on using as many of these services as possible.

1. Introduction

FIWARE is an initiative to provide a platform and a set of standardized APIs to support the creation of Smart Applications in various fields. It initially started in 2011 as an EU's Seventh Framework Programme (FP7) project with the goal of "introducing an innovative infrastructure for cost-effective creation and delivery of services, providing high QoS and security guarantees"[1], having a budget of close to 70 million Euro. Since then it has got significant attention resulting in various follow-up projects[2][3][4]. Besides this, the so called FIWARE Foundation[5] was recently founded with the goal to build a sustainable community around the project. FIWARE is also promoted as a perfect open-source choice for building smart city applications that should prevent vendor lock-in situations.

This paper therefore reports on a feasibility study that was conducted in order to find out how well-suited the FIWARE platform is to support smart city applications. It was actually conducted in two steps. In a first pre-pilot phase the individual components of the platform were analyzed. Based on the findings of this study – especially focusing on those points that were thought to be problematic at that time – a small pilot project was designed to demonstrate the platform's capabilities in a realistic scenario and use-case.

2. The FIWARE Platform

The core of the FIWARE ecosystem is the so called FIWARE platform. It is a set of public and free-to-use API specifications that come along with open source reference implementations. There also exists an initiative called *FIWARE lab*, which offers the platform in a cloud environment. Whereas the *FIWARE lab* is merely for testing, experimenting and evaluation, the *FIWARE iHub* initiative is supposed to provide production-ready cloud services in the future.

The FIWARE platform is grouped in seven major parts called the "generic enablers (GEs)"[6]. Every GE represents a certain aspect of FIWARE services and also provides one or more components along with reference implementations that support the specified APIs. Additionally, there are so called "domain specific enablers (DSEs) that (will) provide components for certain domains like health, energy and so on. The general enablers are organized as follows:

- **Data/Context Management**: This contains all components that are needed to store, access, process and analyze data as part of a smart application
- **Internet of Things (IoT) Services Enablement**: Here are all components needed to setup sensor networks and routing sensor data to other GEs.
- **Advanced Web-based User Interface**: Components to design user interfaces, including geographical information and interactive 3D charts
- **Security**: Components to add, define and enforce declarative security
- **Advanced middleware and interfaces to Network and Devices**
- **Applications/Services and Data Delivery**: Components and tools for data visualization, easy generation of mashups and app-store-like distribution of services and data
- **Cloud Hosting**: Components and tools aiming at providing and managing FIWARE services via cloud infrastructure

URI: http://hdl.handle.net/10125/50615
ISBN: 978-0-9981331-1-9

FIWARE used a great variety of different programming languages (C++, Java, Python, NodeJS, ...) and environments for developing their reference implementations. Fortunately, the FIWARE community provides docker[7] images for every component, which makes dealing with different runtime requirements relatively easy.

3. The Pilot Project

As already pointed out previously, the feasibility study was conducted in two steps. The first one was about analyzing the functionality of the individual components provided by FIWARE without any specific scenario in mind. In order to support a detailed analysis, it was decided not to use the FIWARE lab infrastructure, but to run all components on premise. This also allowed to use the latest version of every component and also to get some insight into the setup and the interaction of these components. One important outcome of this first analysis was, that apparently due to the individual progress of certain components the interaction with other components, which is vital for the operation of the entire FIWARE platform, was no longer working without problems. This seemed specifically true for a key aspect of FIWARE: Security. Thus, the focus of the pilot project was put on implementing a permission system using FIWARE's internal security mechanism, since security is definitely a key issue in smart city applications and therefore a knock-out criterion if it cannot be met.

The context of the pilot project was "smart living". It was not important to demonstrate a lot of functionality, but to point out where an application can benefit from using FIWARE's services rather than creating a proprietary implementation for them. Thus, the goal was to maximize the use of platform services on the basis of a proof of concept. The assumption – actually inspired by a real project – was that there is a new neighborhood to be created on the grounds of a former industrial compound. The newly created apartment and office buildings will be equipped with a variety of sensors that shall be utilized for smart applications. Since the main focus was on security, it was decided to use only one sensor type (temperature) in the pilot project.

The scenario was as follows: A smart mobile application will be given to all tenants living in the new apartment buildings. In every apartment, there is (at least) one sensor measuring the temperature and reporting it to the smart application platform. All tenants can observe the temperature in their apartment(s), including changes over the last five days. They will automatically receive a notification if something extraordinary (e.g. temperature is too low or raises extremely fast) seems to happen. Additionally, there also exists maintenance staff responsible for one or several buildings. They will have access to all sensor data in their building, including those installed in staircases and corridors. If something happens that indicates an incidence or malfunction, responsible maintenance personnel shall also receive notifications. Additionally, the whole system is administrated by a so-called building administration that creates users (tenants, janitors, ...) and registers them with apartments and buildings. It is important that all users have only access to data within their area of interest. Besides this, also sensors are seen as critical infrastructure and therefore must not expose vulnerabilities to the system.

4. The Data Layer

The application needs a mechanism to store its data (buildings, apartments, tenants, …) and usually this data layer is provided by a database. In case of FIWARE there is a service called *context broker*. The context broker is essentially a REST API based on the Open Mobile Alliance's *Next Generation Service Interface* (NGSI)[7]. It comes with a reference implementation called *Orion*[1], which technically consists of a MongoDB database with an NGSI REST API on top of it. It allows for the creation of all necessary entities and does not require any database schema. All entities are stored in a normalized way in one MongoDB collection. Every entity has at least two fields to identify it. One is called 'type' (e.g. "Apartment") and the other one is called 'id' (e.g. "Top12").

The context broker allows for multiple tenants. By using a header-field called "Fiware-Service", that can be unique within one application or domain, data can be nicely separated. Technically the "Fiware-Service" header identifies a MongoDB database used behind the scenes. Thus, if applications use different values for the "Fiware-Service" header field, their requests can never interfere.

The context broker supports all required CRUD operations. By default, however, all requests are based on the so called "normalized" notation, which means that every query returns its result along with metadata like field type. But there exists an option called *'keyValues',* which switches to plain JSON and therefore makes the NGSI protocol transparent to the client application. This greatly facilitates the creation of applications, since they do not have to tread context broker requests any different than other REST APIs.

On the other side, there is also a minor, yet important design flaw in one specific aspect. Similar to the header field "*Fiware-Service*" that can be used to separate the data storage for different applications or domains, there is an additional header-field called

[1] https://github.com/telefonicaid/fiware-orion

"Servicepath" that is intended to further structure or classify data within an application or domain[9]. The problem with using this field is that it cannot be read via the API. Thus, if this field was set, there is no way to later figure out its value. This is critical, since besides *id* and *type* also the *servicepath* is part of the "primary key" that uniquely identifies an entity. Thus, there can be two entities with the same *type* and *id* but different *servicepath*, which leads to an error when making a query by id. Since, however, it is impossible to figure out what the *servicepath* of an existing entity is, this can lead to serious issues. So, although the idea behind the *servicepath* is nice, it should not be used until the API is extended to also include this field in the response.

One of the most powerful features of the context broker, however, is the capability to subscribe to events. This allows to nicely react to changes in the data store. Subscriptions can be of different scope, like "inform me whenever a new entity is created" or "inform me, whenever the temperature in room 212 exceeds 25 degrees Celsius". This greatly facilitates the implementation of smart applications.

In our pilot project, we ended up with the following entity types:

- *Person*: Representing the properties of persons involved in the use-cases, including tenants (living in an apartment), maintenance staff (responsible for one or more buildings), and building administration staff (administering a group of buildings)
- *Building Administration*: A company that owns and/or administrates a set of buildings.
- *Building*
- *Apartment*: Part of a building rented by one or more tenants.

5. Attaching Sensors

An important part of smart applications is the ability to automatically react to changes in the environment. Therefore, sensor networks play a crucial role. This could be surveillance cameras that are used to detect free parking slots and route users to this place, or simple sensors that deliver a single numeric value like the current temperature. Within the pilot project it was important to demonstrate how sensors can be integrated into the platform and how their values are made available to the application on top of it. In our context, we had temperature sensors that were installed in buildings and in apartments. For the sake of simplicity, we decided to have only one sensor per apartment and building, since the focus was less on a realistic scenario but on the interaction of the various FIWARE components. It turned out that FIWARE provides a very nice way to integrate sensors into the application via its Internet of Things (IoT) General Enabler (GE). The reference implementation of this GE is called IDAS[10]. It provides a REST endpoint with the API required for registering sensors and dealing with their data. Before a sensor can be added to the system, in a first step a so-called *service* needs to be created, which serves as the logical endpoint for a group of sensors. The general idea is that sensors provide values for properties of entities stored in the context broker (e.g. the temperature of a specific room or apartment). So, when registering a new sensor device, a reference to this entity needs to be set up. This reference has to include the *entity_name* (which is the *id* property within the context broker), the *entity_type* (e.g. 'building', 'apartment', …) and the name and type of the entity's property that will eventually hold the sensor's value. The registration also has to include a *device_id* that is used to uniquely identify the sensor.

Thus, whenever a sensor reports a new value to the IoT component via REST (using its *device_id* and the *id* of the IoT service), the value is extracted from the request and the corresponding entity within the context broker gets updated by the IoT component. So, in our example every apartment and building had a property called temperature that was always holding the latest value reported by the connected sensor.

Here the problem with the *servicepath* property described in the previous chapter became evident. The IoT component can only identify a specific entity using a pair consisting of *entity_name* and *entity_type*. The context broker, however, internally uses a triple consisting of these two fields and the *servicepath*. So, if the *servicepath* of an entity is set to some non-default value, it can no longer be referenced by IDAS. In such a case, instead of updating the existing entity, a new entity with the given *entity_name, entity_type* and the default *servicepath* is created that only holds the property defined by the sensor (temperature in this case).

Another important fact is that with every new sensor value, the previous value gets overwritten. Thus, there is no time-series of values stored in the context broker.

6. Historic Data and Time Series

When only using the IoT and Context Broker components, no time series data will be available, since new sensor data replaces the existing one. If, however, the historic data shall be preserved, an additional component called Cygnus[11] is required. This component is essentially an extension of Apache Flume[12].

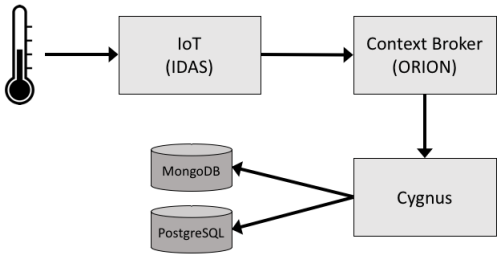

Figure 1: Data flow upon arrival of new sensor value

The idea is to create a subscription with the context broker in order to get informed once a particular property of the entities of a specific type (e.g. the *temperature* property of entities of type *apartment*) changes. Whenever such an event occurs, the data related to this event gets stored in a data sink. Possible data sinks are MongoDB, HDFS, PostgreSQL and many more. This data and event flow is shown in Figure 1. When the sensor sends a new value to the IoT component, this will result in an update request sent to the context broker, which in turn causes the context broker to inform all interested subscribers. Cygnus receives the event sent from the context broker and stores it in the registered data sink(s).

This architecture allows for a clear separation of live data stored in the context broker and the historical data stored in any database of choice. Having split the task over several loosely and asynchronously connected components allows for high performance and throughput. Probably most important, this can all be achieved without a single line of programming so far.

Since everything in FIWARE is about REST-based APIs, there is also a component that allows for RESTful access to the historic data sink. The name of this component is Short Term Historic (STH) and the reference implementation is called Comet[13]. It provides an API for reading historic data produced by the component chain described above, but only supports MongoDB data sinks so far.

7. Complex Event Processing

To make our pilot project "smart", the goal was to automatically recognize suspicious patterns in sensor values that might indicate incidents or malfunction. As was already mentioned in section 4, simple cases (e.g. temperature lies outside a specified range) can be solved solely relying on the context broker's subscription model. But what about a sensor that is defective and does not send a signal at all? Or a sensor that does report a rapid temperature increase by more than 10°C within 10 minutes?

To recognize things like this, the FIWARE platform provides the so-called Complex Event Processing (CEP) General Enabler with a reference implementation called Proactive Technology Online (Proton)[14].

During the evaluation, it became quickly clear that this implementation is far from being production ready when it comes to usability and error handling. While generally the documentation of the FIWARE components seems not always to reflect the latest version of the actual software, the documentation on Proton is specifically poor and it took lots of trial and error and research on *stackoverflow* to get it running at all. For example, Proton supports different types of events. They are called *ContextUpdate* (which means incoming data from sensors, typically via context broker subscriptions), *Alert* and *Warning* (which both stand for outgoing events). However, when creating a new event there is no way to define its type other than using a naming "convention". This is not actually a "convention" since failing to name events with exactly these suffixes results in an error. The error message, however, does not tell the user about this "convention" but is a plain Java NullPointerException stack trace with no clue about the reason of the problem at all. Consequently, Proton must be considered an "expert system" that apparently can only be used by people who have been closely involved in its development process.

Anyway, with this tool, it is possible to create time-framed rules via so called *TemporalContexts*. This way it was possible to react to a certain number of sensor events within a given period of time. Unfortunately, we failed to distinguish between individual devices, thus, all events coming from any sensor were considered here. The documentation describes a so-called *SegmentationContext* that is designed to keep different event sources separated. But since the time budget reserved for the evaluation of this component was already exhausted, we had to give up on that. Even tutorials on the web[2] mention that there are still severe bugs in the software. In response to issue reports on Github, we learned that development of this component was already stopped. Thus, this part of the platform needs to be considered incomplete and it is more likely the better option to integrate some alternative tool instead (e.g. Apache Flink's CEP[15]).

8. Security

As we already pointed out in section 3, security was a key requirement for the pilot project. In fact,

[2] https://appshelfer.de/09/

none of the components that have been discussed so far provides any security mechanism at all. This means, that as soon as the context broker is up and running there are no restrictions on using the REST API. As a consequence, every user can read, write and also delete any data stored there. The same is true for all other components. So how can these resources be protected then? Definitely the first thing that needs to happen is that none of the REST endpoints must be accessible from any untrusted network. For the rest, the FIWARE platform provides three components that need to interact together in order to provide controlled and safe interaction with the other services and applications. These components are:

- Identity Manager (IdM): This is a service to create users, roles and permission.
- Policy Decision Point (PDP): This service provides authorization by deciding whether the current user is allowed to perform a certain action
- Policy Enforcement Point (PEP): This is a proxy server that performs the actual authentication and optional authorization checks in interaction with the other two components

8.1. The Identity Manager (IdM)

The IdM is the central component of the FIWARE security architecture. Its reference implementation is called Keyrock[16] and it is based on OpenStack Keystone[17], which in turn is an open source implementation of the OpenStack Identity API[18]. Keyrock is – besides the CEP rule editor – the only FIWARE component that comes with a web-based user interface. This web interface is internally called *Horizon* whereas *Keyrock* more specifically refers to the REST interface.

Keyrock is essentially an OAuth2[18] authorization server and therefore supports authentication for the entire platform as well as client applications on top of the FIWARE infrastructure. It is holding all user information and is a single sign-on service for all components and applications. Thus, applications do not necessarily need to maintain user information (especially no private credentials) and one account can be used for all applications using the platform.

With the IdM's web interface, it is possible to create/register:

- Users ("resource owners" according to OAuth2) with their credentials
- Applications using FIWARE services ("clients") along with an automatically generated pair of credentials needed to interact with the authorization server (e.g. to retrieve the access token)
- Roles within an application (logical names of different user groups)
- Permissions (detailed access rules) assigned to roles
- Assignments of roles to users in the context of an application

8.2. Policy Enforcement Point (PEP)

The reference implementation of this component is called Wilma[20] and it is playing the role of the so-called *resource server* according to OAuth2. The resource server is hosting information that is sensitive and therefore can only be accessed by authorized requests. In fact, sensitive information is stored in the context broker, the IoT service and of course also in our client application and several other services. Wilma is therefore implemented as a simple proxy server (also called PEP proxy) that is adding security by transparently acting on behalf of the actual service that needs to be protected. Thus, instead of allowing direct access to a sensitive service, clients interact with the proxy (see Figure 2). It is also possible to configure so-called public URLs for which the proxy won't perform any security checks. Otherwise the proxy checks authentication with the other security components (in this case the IdM) and forwards the request to the actual resource server (called "Back-end Apps" in the graphic) if security constraints are met. Therefore, the PEP proxy needs to know about the addresses and ports of the service it is protecting and of the other security components. Additionally, when registering a new application with the IdM, also a pair of credentials for the PEP proxy is generated that is needed by the proxy to authenticate with the IdM.

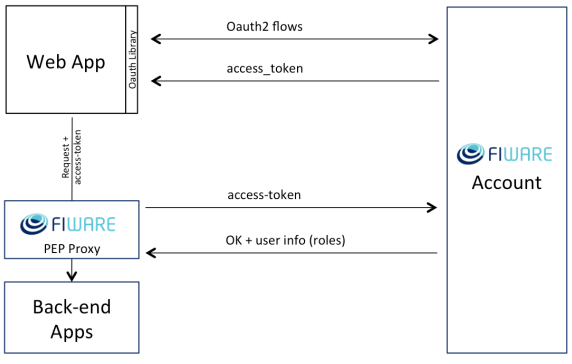

Figure 2: Authentication (Level 1 Security)[20]

So, if the request is properly authenticated, the protected application has access to the user id and the roles of the current user, which are provided as a list of role names. It is important to realize that in this scenario (called Level 1 security in FIWARE) only authentication is checked, but it is not tested, whether the user is allowed to perform the current action. To deploy authorization, there exist two possible options.

- Application layer security
- Platform layer security (Level 2 or Level 3 Security)

In the first case, authorization decisions are made as part of the application logic. Since the application has access to the current user's id and roles and it also knows about the semantics of these roles in can decide whether it allows for this action to be performed or not. Alternatively, also the FIWARE platform supports authentication based on so called permissions (see next section), which takes security related decisions out of the application logic and performs them on the platform layer.

FIWARE provides a sample application[3] that is designed as a tutorial for using FIWARE's GEs. This tutorial is using application layer security and therefore cannot be used as a reference to the Level 2/3 security approach. Since the goal of our pilot was the demonstrate the platform services (with a clear focus on security), we decided to go for platform authorization.

8.3. Policy Decision Point (PDP)

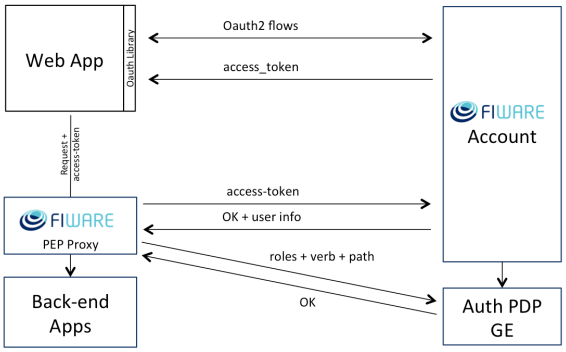

Figure 3: Platform level authorization (Level 2)[20]

When making use of platform level authorization (called Level 2 and Level 3) an additional component called the Policy Decision Point (PDP) is needed. FIWARE provides a reference implementation called AuthZForce[21]. The authorization flow is shown in Figure 3.

[3] https://github.com/Fiware/tutorials.TourGuide-App

In this scenario, the PEP – after having checked the validity of the access-token with the IdM – makes a consecutive request to the PDP providing the current user's roles and the request details (URL plus the HTTP request method used). The PDP checks this information with its security policies and decides whether access should be granted or not.

Access rules – called permissions in FIWARE – are written in the eXtensible Access Control Markup Language (XACML)[22]. Up to Level 2 security, developers do not have to deal with the details of XACML. When defining a permission, which always is part of a specific role, the IdM's user interface simply accepts a URL and a http verb (e.g. GET, POST, PUT, …) that should be granted to all users belonging to the corresponding role. The necessary XACML is created automatically behind the scenes. Level 3, however, allows users to write custom rules using XML. Although rules are edited using the IdM web interface, they are required to be stored in the PDP. Thus, every time a user-profile or a role gets updated, the corresponding XACML policies are re-generated and automatically transmitted to the PDP via REST. This, however, requires access rules to be stored redundantly within the IdM and the PDP, which turned out to be rather problematic during the evaluation. Besides this, every change to a role or user results in a new XACML security policy that is then made the active one, while the older versions are still kept. However, the number of policies per rule within AuthZForce is restricted to ten. So, after 10 changes there will be an error, that can only be solved by deleting older policy versions via REST requests.

8.4. Security and the pilot project

As already pointed out, it was decided to use (at least) level 2 security in order to evaluate platform level authorization. One of the potential benefits was to entirely exclude security from the application and to use a declarative rather than programmatic concept.

One of the first problems was to make the three components (IdM, PEP and PDP) work together. It turned out that the documentation on how to configure this interaction is rather scarce. Besides this there had been some conflicts between the then available docker images that got solved by the FIWARE components' development teams after a couple of issue reports via GitHub.

Besides these problems, it soon became clear that a clean separation of security concerns between the application and the platform was not possible. Inside the application all persons (tenants, janitors, ...) and therefore potential users are stored as entities in the context broker. One use-case was to list the current temperature of all units the current user has access to.

This requires establishing a mapping between the concept of a user as it is kept within the application and the user as defined within the IdM. While this was simply solved by using the same user name on either side, some other problems resulted in a significant programming effort. For example, whenever a tenant was assigned to an apartment, this required to set up the appropriate access rights allowing this user to read the respective data of the apartment, including of course the current temperature. To keep this in sync manually by requiring the administrator to maintain data about the user using our pilot project's interface while simultaneously setting up the proper permissions using the IdM interface is completely infeasible. Thus, the decision was made to make all changes to the security configuration from within the application. So, every time a new user (tenant or maintenance staff) is created in the application a corresponding user is created in the IdM using the REST API. Also, when a user is made a tenant of an apartment (or a janitor of a building) the proper roles along with their permissions are also automatically created using REST. Here the aforementioned redundancy and a lack of functionality in the IdM's API became problematic. It is relatively straight forward to define permissions in the IdM's web interface. When saving these changes, permissions are internally translated into XACML and sent to the PDP via its API. It is also possible to create the same roles and permissions via the IdM's REST interface instead of using the web user interface, but there is no way to cause the IdM to transfer these programmatically made changes to the PDP. Consequently, it was necessary to directly send REST requests containing the correct XACML to the PDP. This would have allowed to write more complex rules than supported by the level 2 security mechanism. Unfortunately, it turned out that whenever the IdM's web interface was used to make any change to the security configuration it caused the IdM to generate a new XACML policy based on its local configuration and to send this to the PDP. This will cause the programmatically – and potentially more sophisticated – configuration made by our pilot application to be overwritten by the IdM. Therefore, the pilot application was limited to the same rules that are generated by the IdM and it was decided to create these rules redundantly within the IdM and the PDP. This way, even when the IdM overwrites a policy, the result is essentially the same.

Thus, in the end it turned out that a good part of the logic that is already part of the IdM (e.g. creating XACML policies out of permissions and sending them to the PDP) had to be re-implemented as part of our application. On the other hand, some shortcomings of the IdM had to be solved as well, like deleting older policies once they are no longer used. So, it is really questionable whether using application layer security in the first place would not have been the better option.

While – although with significant effort – the application could be secured properly against unauthorized access from the client side, also sensors are forming a massive security risk, since potential attackers can get easily access to their hardware. Thus, while protecting the communication channel (e.g. with TLS[23] and/or VPL[24]) is important, it is especially critical to enforce authentication and authorization here as well. The IdM allows to create credentials for sensors as well, which is very similar to registering a PEP. These credentials need to be used by the sensor, once access to the IoT service is protected using a PEP proxy. Unfortunately, it turned out that the credentials produced be the IdM's web interface – due to some restrictions in the underlying user type that gets created within KeyRock - can only be used for level 1 security only. This means that there is no authorization possible and that every authenticated sensor can technically send values on behalf of any other known sensor. This would allow attackers to use this sensor account to get potentially unlimited access to the sensor network. The solution to this unacceptable security hole was to use standard user accounts instead of the special sensor accounts. Once again, these accounts are automatically generated by the pilot application whenever the administrator adds a sensor to an apartment or building. This will also create a permission rule that allows this sensor to only send data of a specific type to a specific service endpoint eliminating the chance to spoof some other sensor's identity.

9. The big picture

In Figure 4 the architecture of the entire pilot application and its interaction with the FIWARE components used is shown.

Every labeled box represents a docker container and can therefore be seen as a virtual machine. The big box indicates the trusted environment without any security restrictions. The box labeled "FSH Server" represents the server-side logic of our pilot application. On the top, we see the mobile client application that is given to all tenants and maintenance staff. The admin application is used by the building administration to manage buildings, staff and tenants. At the lower end the sensor network is indicated. All external clients need to use a PEP proxy for accessing services.

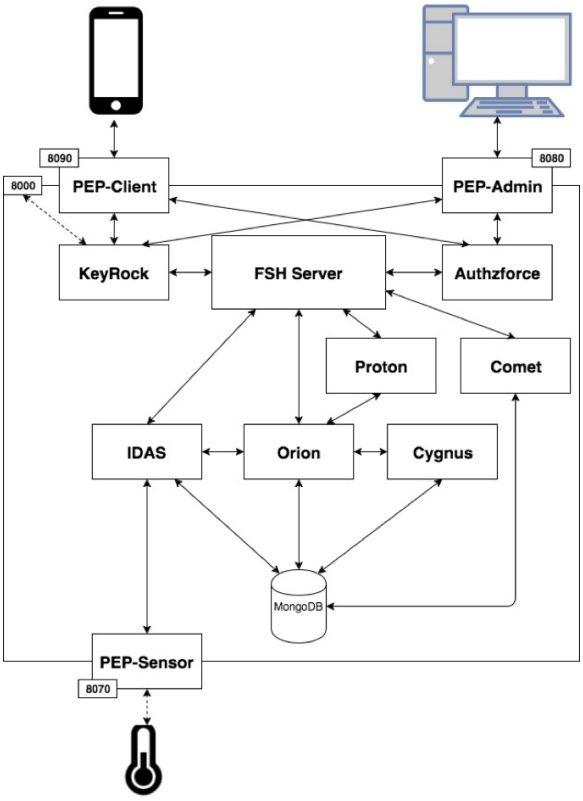

Figure 4: The overall system architecture

This makes sure that all requests are authenticated and authorized. So, once a request has passed a PEP proxy the application can trust it. Beside the PEP proxies also the port of the Keyrock service is publicly available. This is necessary to enable OAuth2 authentication. The server-side application logic interacts with various FIWARE services. Keyrock and AuthZforce are used to create accounts, roles and permissions. IDAS is used to register sensor devices and to route sensor data to the correct building or apartment. Orion (the context broker) is used to hold the application's domain. Proton is used to register subscriptions with the context broker and to emit alerts once some unusual pattern in sensor data is detected, while Comet is used to read the temperature history of a sensor (see screenshot in Figure 5).

One constraint that was initially perceived to be rather limiting is the fact that every application can contain only one PEP proxy and that every PEP proxy can only protect one service endpoint. For this scenario, however, three PEP proxies have been necessary, leading to three logical applications. This turned out to be very intuitive and helpful in the end. Although the pilot was considered to be one application and the server-side logic is just one application – thanks to this PEP proxy constraint – we ended up with the following three 'logical' applications registered with the IdM:

- The Client Application
- The Administrative Application
- The Sensor Application

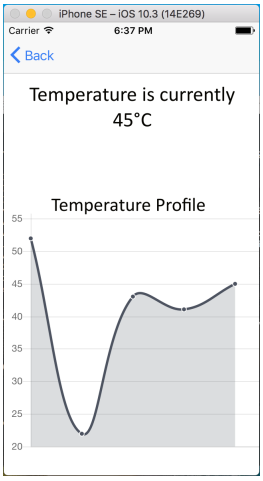

Figure 5: Screenshot of the mobile client application

The practical implications are that all roles, permissions and their assignments to users are separated from one another. Thus, we have an individual set of users, rules and permissions for every of these logical applications, which leads to a clear structure that is considered to be very helpful.

10. Conclusions

The overall goal of this case study was to get an unbiased analysis of the capabilities of the FIWARE platform from a software engineering point of view. It was important to find out how and to which extend smart applications can benefit from making use of the various services provided by the platform. In the end, it could be demonstrated that there is actually some point in building on top of FIWARE, since there is a huge set of functionalities that supports smart application development. However, it also needs to be said, that the way getting there was really tedious. This had partly to do with the fact that FIWARE is a really large and complex platform with a huge set of components and services, which definitely takes some time to learn and understand, but had also to do with the lack of up-to-date documentation and some severe bugs as well as several design flaws that still exist in some of the components. This was especially surprising since already the initial effort to come up with such a platform was funded with close to 70 million Euros.

As it could be demonstrated with the pilot project, smart applications can significantly benefit from using the FIWARE platform. As long as the *servicepath*

property is not used (see section 4), the context broker provides a stable and reliable persistence layer. Especially its subscription mechanism greatly helps to meet typical smart application requirements without the need of programming. The same is true for FIWARE's IoT component and the idea of keeping live data separated from historic data. However, it needs to be stated that no performance and load tests have been conducted. Concerning the pilot's security implementation, retrospectively it might have been less effort to use application layer security, allowing the application to make authorization decisions programmatically. Consequently, this is a design decision that needs to be made carefully, considering all sorts of implications like the level of trust between client-applications and platform components. But also some really severe issues have been identified during evaluation. Probably the biggest problem is the extremely poor quality of the CEP component Proton, which needs to be considered a key component for "smart" applications. As described in section 7 this component is extremely hard to use and contains several serious bugs. Even worse, this project was apparently stopped and is no longer under development. Thus, if complex event processing is needed – which is very likely in a smart application – alternative solutions need to be found. If the currently shaping FIWARE foundation really wants to push the platform, some significant action is required here. A second huge problem is the web interface of the IdM. Some of the issues here have already been discussed in section 8. But additionally, the interface falls short on some very basic functionalities. For example, it is currently not possible to modify, delete or even view existing permissions, which does not make the application fit for real-life use. On the other side, it could be demonstrated that there is always a workaround possible. So, if several organizations decide to use the FIWARE stack and commit their solutions and bug-fixes back to the public repository a lively and self-sustaining open source community could emerge that would make the whole platform even more attractive. Definitely the biggest advantage of the FIWARE idea is to use open standards that are designed to avoid vendor lock-in situations.

11. References

[1] Publications Office of the European Union, 2011, *FI-WARE: Future Internet Core Platform*, EU, Brussels, Belgium

[2] Publications Office of the European Union, 2016, *FI-GLOBAL: Building and supporting a global open community of FIWARE innovators and users*, EU, Brussels, Belgium

[3] Publications Office of the European Union, 2017, *A FIWARE-based SDK for developing Smart Applications*, EU, Brussels, Belgium

[4] Publications Office of the European Union, 2017, *Bringing FIWARE to the NEXT step*, EU, Brussels, Belgium.

[5] FIWARE, 2017, https://www.fiware.org/foundation/

[6] FIWARE 2017, *The FIWARE Catalogue*, https://catalogue.fiware.org/.

[7] John Fink, 2014, *Docker: a Software as a Service, Operating System-Level Virtualization Framework*, code{4}lib Journal, Issue 25, 21.04.2014

[8] Open Mobile Alliance, 2012, *NGSI Context Management, http://www.openmobilealliance.org/release/NGSI/V1_0-20120529-A/OMA-TS-NGSI_Context_Management-V1_0-20120529-A.pdf.*

[9] FIWARE Orion Team, 2017, *FIWARE-ORION Documantation – Entity service paths*, http://fiware-orion.readthedocs.io/en/master/user/service_path/index.html

[10] Carlos Ralli Ucendo, 2016, *Backend Device Management – IDAS*, https://catalogue.fiware.org/enablers/backend-device-management-idas

[11] FIWARE Cygnus Team, 2017, *Cygnus*, http://fiware-cygnus.readthedocs.io/en/1.2.2/index.html

[12] Apache Flume Team, 2017, *Apache Flume User Guide*, https://flume.apache.org/FlumeUserGuide.html

[13] FIWARE Comet Team, 2016, *FIWARE Short Time Historic (STH) - Comet documentation* ,https://fiware-sth-comet.readthedocs.io/en/latest/

[14] Uri Shani, 2016, *Proton Documentation* https://github.com/ishkin/Proton/blob/master/documentation/Readme.md

[15] Till Rohrmann, 2016, Introducing Complex Event Processing (CEP) with Apache Flink https://flink.apache.org/news/2016/04/06/cep-monitoring.html

[16] Joaquín Salvachúa and Álvaro Alonso, 2016, *Identity Management – KeyRock*, https://catalogue.fiware.org/enablers/identity-management-keyrock

[17] The OpenStack Foundation, 2017, *Keystone, the OpenStack Identity Service*, https://docs.openstack.org/developer/keystone/

[18] The OpenStack Foundation, 2017, *Identity API v3* https://developer.openstack.org/api-ref/identity/v3/

[19] D. Hardt (Ed.), 2012, *The OAuth 2.0 Authorization Framework*, https://tools.ietf.org/html/rfc6749

[20] Álvaro Alonso, 2016, PEP Proxy – Wilma, http://fiware-pep-proxy.readthedocs.io/en/latest/

[21] Cyril Dangerville, 2017, Authorization PDP - AuthZforce, https://catalogue.fiware.org/enablers/authorization-pdp-authzforce

[22] Erik Rissanen (Ed.), 2013, eXtensible Access Control Markup Language (XACML) Version 3.0, http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html

[23] T. Dierks, 2008, *The Transport Layer Security (TLS) Protocol, Version 1.2*, Internet Engineering Task Force – Network Working Group, RFC 5246, https://tools.ietf.org/html/rfc5246

[24] E. Rosen, Y. Rekhter, 1999, *BGP/MPLS VPNs*, Internet Engineering Task Force – Network Working Group, http://www.ietf.org/rfc/rfc2547.txt