Analysing FIWARE's Platform - Potential Improvements

Peter Detzner
Automation and Embedded Systems
Fraunhofer Institute for Material Flow and Logistics
Dortmund, Germany
peter.detzner@iml.fraunhofer.de

Peter Salhofer
Information Management
FH JOANNEUM
University of Applied Sciences
Graz, Austria
peter.salhofer@fh-joanneum.at

Abstract

The digitalization of the world proceeds continually. As part of this, we are facing a smart digital future where everything is connected and linked while we are transforming big data into knowledge. The FIWARE platform, which is the result of a series of EU funded projects, supports this transformation. Developers, integrators and end users are able to rapidly create new applications, by using components provided by this platform. We have analyzed central building blocks of the FIWARE platform from different aspects and - based on experiments and code reviews - like the need for a multi-level role-based security or a caching mechanism when it comes to querying data. Within this paper we also provide further recommendations for the current code base that can help to improve those FIWARE components under investigation. In addition to that we also suggest how to simplify a typical setup by extending FIWARE's core component using a Websocket-Port.

1. Introduction

The goal of FIWARE is to provide a standardized platform for smart (city) application development. It was initially started in the year 2011 and the entire initiative was reasonably funded by a sequence of EU projects[1, 2, 3, 4]. Its technical core is a set of RESTful APIs[5] that come with reference implementations and are all open-source. FIWARE is currently intensively promoted by organizations like the FIWARE Foundation[6]. The authors of this paper worked independently of each other in different FIWARE-based projects and have gained significant practical experience. Although the FIWARE platform provides lots of advantages (e.g it is open-source and its components are usually well-integrated) we came across several things that could or should be improved. Thus, the goal of this paper is to point out these weaknesses and also to suggest potential improvements to these shortcomings.

The paper is structured the following way: We will provide an overview of the core-components of FIWARE and how these components are used in a typical IoT scenario. This introduction will also briefly describe the functionality of each of these components. In the following chapters we will point out our findings of potential weaknesses and how they could be improved. Finally, we will summarize our results in the conclusions chapter.

2. A FIWARE Primer

Figure 1 shows a simple IoT application built out of FIWARE components which usually communicate via HTTP REST[7]. A central building block is the so-called context broker, represented by its reference implementation called Orion[8]. Technically this is a Mongo NoSQL database with a REST API based on the Open Mobile Alliance‘s Next Generation Service Interface (NGSI) protocol[9]. The context broker represents a persistent storage that holds the current state of the application. Besides storing data other components can subscribe for change-events on certain types or attributes of entities. All subscribers will be notified by the context broker in case that a relevant element got changed. Orion also provides the concept of multi-tenancy. Different tenants or applications can be separated by a custom HTTP header field called `Fiware-Service`. As a result, the context broker creates a separate MongoDB database for every Fiware-Service name used.

Another important component in the FIWARE platform is the IoT agent IDAS[10], which takes care of the inbound traffic from sensor networks. Before a sensor can send data to the application two things need to be done upfront:

1. A so-called IDAS Service needs to be created that logically represents a group of sensors and results in a so-called API-key
2. Every sensor needs to be registered with IDAS, resulting in a device id.

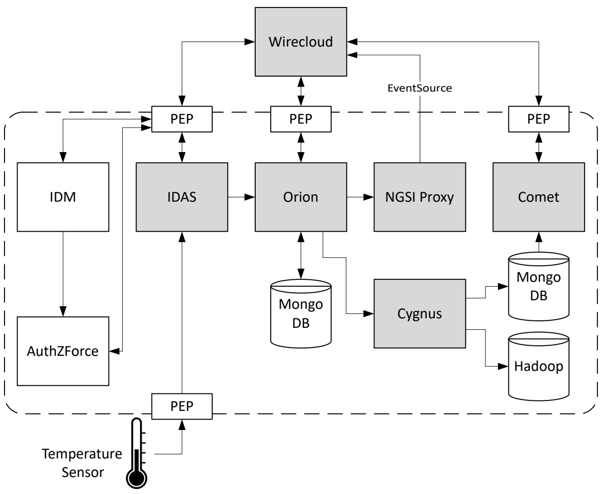


Figure 1. A typical setup of FIWARE's core components. Grey: Generic Enabler in FIWARE platform, white: Security components

The actual sensor needs to send the API-key and the device id with every request. When registering a sensor, it is necessary to define which data it transmits (e.g. temperature, air-pressure and humidity) and to which entity and which attribute(s) this data should be sent. In case that no such entity exists in the context broker, it will be created upon the first transmission of sensor data.

Let's use a simple example to illustrate this: We are developing a smart living application for an apartment building. Thus, our business model stored in the context broker will hold data of all the apartments in the building and the rooms that belong to an apartment. We can now define temperature sensor *T42* and declare that its value should be stored as the *temperature* attribute of the entity with id *App13Room3*. Consequently, every time sensor *T42* sends a new value, IDAS will create an update request for the context broker in order to set the *temperature* attribute of entity *App13Room3* to this value. This ensures that the context broker always holds the latest sensor value. However, every time a value in Orion gets changed, the old value will be overwritten.

If we are interested in keeping the historical data as well, we need to use another component called Cygnus[1]. This component is responsible for recording time series data. Therefore, it uses Orion's subscription mechanism. As already pointed out previously, each subscription can either be made for an entire category of entities or for a specific one. It also has to include all attributes that should be tracked for changes and transmitted to the subscriber whenever a change occurs. Cygnus, which is based on Apache Flume[2] uses so called channels to forward the data it receives to various data sinks. There exists a great variety of channels (e.g. Mongo[3], Hadoop HDFS[4], ...). It is possible to combine these channels so that data is written to several sinks simultaneously.

In order to provide RESTful access to the historic data, there exists a component called Comet[5]. Whereas Cygnus is able to write data to various sinks, Comet can only read them from a MongoDB data sink. The Comet API allows for querying time series for a specific attribute within a given period of time.

None of the components that have been presented so far implement any security mechanisms. Thus, once there exists access to any of them, there are no restrictions on what could be done (including deleting all entities for instance). Security in FIWARE is entirely based on the OAuth2 protocol[11] and consists of three different components that are orthogonal to all the other ones. The central component is called Keyrock or simply the identity manager (IdM)[12]. It plays the role of an OAuth authorization server and provides user accounts for all FIWARE based applications. These applications can register with the IdM similarly to other OAuth-based services like Github, Google or Facebook. Once an application is registered with the IdM, roles for these applications can be specified and assigned to users. The second important component when it comes to security is called the Policy Enforcement Point (PEP) with its reference implementation called WILMA[13]. The PEP acts as proxy server to the resources it protects. Thus, instead of allowing users or applications to directly access services like Orion, IDAS or Comet, they only get access to the corresponding proxy server. The PEP proxy checks every incoming request for the existence of an authentication token. If no such token exists, the user is redirected to the login page of the IdM. In case that there is such a token, the validity of this token is checked with the IdM. In order to allow the PEP making such a request, each PEP has to login into the IdM with its own credentials that have to be setup upfront. Once verification of the token was successful the PEP adds information about the current user and assigned roles to the request.

What happens next, depends on the configuration of the security infrastructure. In the simplest of all cases verification stops after authentication without any form of authorization. Consequently, once a user or an application got successfully authenticated, there are also no restrictions on what could be done. In this case

[1]`https://fiware-cygnus.readthedocs.io`

[2]`https://flume.apache.org`

[3]`https://www.mongodb.com`

[4]`https://hadoop.apache.org/`

[5]`https://fiware-sth-comet.readthedocs.io`

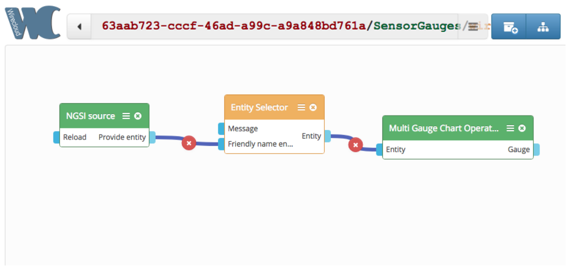


Figure 2. Wiring view of a simple dashboard (Screenshot from one of our projects)

security has to be entirely implemented by a custom application that needs to be put behind the PEP proxy and decides internally on the given user and roles which requests to permit.

In cases where also authorization is needed, another component, called AuthZForce[14], becomes important. When defining application specific roles inside IdM, there exists the possibility to assign so called permissions to each roles. A permission in its simplest form consists of a HTTP method (e.g. GET, POST, DELETE, ...) and an URI describing the endpoint of this request. All requests covered by these permission will be allowed. In a more complex scenario these permissions can be expressed as XACML[15] rules, which are saved and interpreted by AuthZForce. Prior to version 7.x of IdM all authorization, including simple rules, was using AuthZForce. Thus, as soon as authorization was enabled, the PEP proxy made an extra round-trip to the AuthZForce server to determine whether a particular request is allowed. Since IdM version 7.0.0, however, the evaluation of simple authorization rules that do not necessarily require XACML was integrated into the IdM, which avoids the need of an extra round-trip to AuthZForce.

In order to visualize data stored in the system, FIWARE provides a component called *Wirecloud*[16]. It provides a simple mean to build individual dashboards by wiring together different components as shown in Figure 2. Wirecloud comes with a set of such components that fall into two categories:

- **Operators**: These are used to read, process and write data from and to the other FIWARE components
- **Widgets**: Provide a graphical user interface that can be used to present data or to interact with the user

Additional components can be retrieved from the online store that is tightly integrated into Wirecloud or custom components can be programmed and integrated.

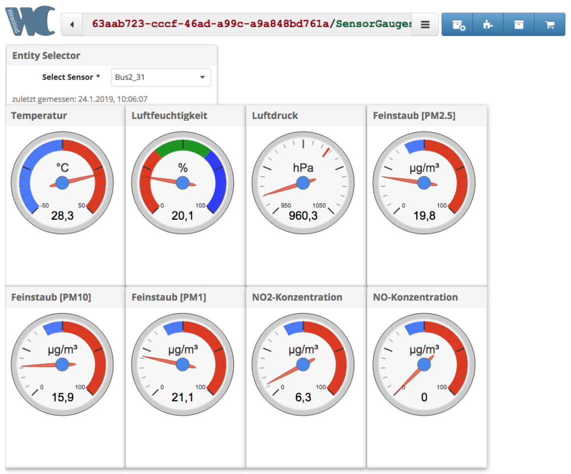


Figure 3. Screenshot of the visual representation of the flow shown in Figure 2

Technically each component is a collection of JavaScript files and assets along with metadata that is uploaded as a zipped archive to the Wirecloud portal. An example of how such a flow is translated into a dashboard can be seen in Figure 3.

Wirecloud consists of a server application that is used to interact with the rest of the FIWARE platform and a JavaScript part that runs locally in the user's browser. In order to constantly receive the latest value for relevant attributes, Wirecloud can register a subscription with Orion. Since Orion cannot directly notify the JavaScript application that runs somewhere in a user's browser, a component called *NGSI Proxy* is used as the subscriber. Additionally Wirecloud registers an *EventSource* with the *NGSI Proxy* to actually receive notifications. This allows for user interfaces that are automatically updated with every new sensor value. This mechanism is also depicted in Figure 1.

3. FIWARE'S ORION CONTEXT BROKER

As already mentioned the most important component inside FIWARE's platform is the so-called context broker Orion. Orion allows with operations like updates, queries, registrations and subscriptions making other applications context aware and managing the entire lifecycle. Publishers of an entity are called *Context Producers* while subscribers of an entity are called *Context Consumers*.

An entity represents a physical or virtual object, like a sensor, a robot or a thing in general. Each entity, as it is depicted in Figure 4, can be uniquely identified by the two fields, the *type* and the *id*. An

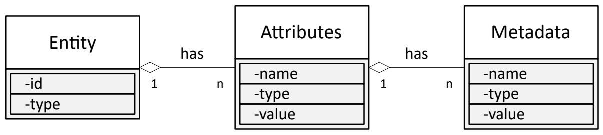


Figure 4. Example of a NGSI v2 Entity

entity *type* represents the type of the thing, whereas the *id* uniquely identifies the entity. Each entity has one or more attributes identified by *name*, *type* and *value*. The attribute *value* holds the actual data, while the *type* represents the value's data type, e.g. JSON datatypes or NGSI types. Each entity attribute can have one or more metadata attributes, also using a metadata *name*, *type* and *value*. The metadata *name* describes the role of this metadata, the *type* represents the value's type and the *value* contains the actual metadata.

3.1. HTTP Header Best Practices

Orion offers a HTTP RESTful API[7][17] for managing context data, like entities. Each HTTP Request has the structure as depicted in the Table 1, where the `Request Line` and the `Header Fields` are mandatory, while the `Body` is optional:

Table 1. HTTP Request

METHOD URI PROTOCOL_VERSION	Request Line
Request Modifiers Client Information Representation metadata	Header Fields
Body	(Optional)

This section analyses the HTTP header fields used, based on the best practices according to [18, 19, 20, 21, 22, 23, 24]. Our presented improvement suggestions for the HTTP headers should help to simplify few aspects but also reduce the communication bandwidth in general.

Deleting an Entity

When deleting an entity, Orion does not accept any payload or even the `Content-Type` for the HTTP header[25]. According to the set of rules for a HTTP header[18], the `Content-Type` must be used and the `Content-Length` should be used. But since the `Content-Length` is 0 in the request, it is questionable why the `Content-Type` is analysed at all. [26] stated that some implementations might reject payload, we propose only to analyse the `Content-Length` and ignore the `Content-Type` in case of deleting an entity which would be also in favour of future developers.

If-Modified-Since inside a Request/ Last-Modified should be used in response

The usage of the *If-Modified-Since* HTTP header could also help to support conditional requests[27]. The client, while sending a GET-request with a defined *If-Modified-Since* attribute, can determine with this field the actuality of its local copy, in case that a subscription is not possible, not necessary or not even needed. The response should be a HTTP status code 304 (*Not Modified*) [28] without any body but including the `Last-Modified` time inside the `Response Header`. In case a modification happened after the `If-Modified-Since` date, the response would have a HTTP status code 200 (`Success`) including the response inside the body. Adding by default the *Last-Modified* could also help to increase the performance of Orion and reduce the network traffic but also to alter the representational state of this entity. To do so, developers have to make sure, that the evaluation of the *If-Modified-Since* header is faster than retrieving data from the database, processing and sending it over the network to the destination. These timestamps, to support these feature, are already foreseen in the database model of Orion by using the `modDate` in MongoDB for example.

ETag should be used in responses

Using `Last-Modified` and `If-Modified-Since` headers in distributed systems as it is foreseen in FIWARE's platform, might create some new problems. This requires that the participants are having synchronized clocks and a common understanding of time. Another challenge might be the transport of data over large distances like the WWW, which requires some time. Providing the `ETag`, which "*is an opaque string that identifies a specific version of the representational state contained in the response entity*"[18]. This `ETag` can be reused by clients for future requests, easily by adding this opaque identifier to the `If-None-Match` field as a part of the HTTP header. Orion would return, in case nothing has changed according to this identifier, the HTTP 304 (`Not Modified`). Orion has only to check the internal `ETag` instead of querying the whole entity, converting the document into a JSON object and sending the data to the client. It would also preserve the bandwidth and time by not sending the complete object. This best practice could be supported by using the already existing `lastCorrelator` of the MongoDBs database model. This field is automatically created by the database driver in Orion to represent a document inside MongoDB, by using the built-in function of

Linux for creation of the universally unique identifier. Combinations of the `If-Modified-Since` and the `If-None-Match` header fields are also possible.

Versioning

Designing a good/clean/proper HTTP REST API including a simple versioning is a very important task, which effects developers of the server as well as developers of client applications. REST has no guidelines for versioning, as a consequence there are many different approaches on how to address it. Orion's versioning is based on the NGSI implemented version, which is represented in the Uniform Resource Identifier (URI). This means sending a request towards the NGSIv1 implementation, the client has to use v1 in the URI, sending requests towards the NGSIv2 implementation must use v2 in the URI. One advantage of having the version encoded in the URI for example is that developers are aware of where they are looking at, especially new developers. The downside of this approach is that with every new version the URI changes.

Another approach to avoid multiple URIs for versioning could be using the `Content-Type` header. This would increase the effort of mapping between one or more APIs at the Orion developer side. Another possible solution could be to create a new custom header like "Version: 1", but we are highly recommend not to do this. We prefer to use already existing, standard HTTP headers instead of introducing new ones which are not specified in the standard. According to [29], nine out of ten APIs are using the URI path for versioning, a more detailed discussion about versioning is covered in [30]. We would appreciate thinking here in favour of the growing developer community.

Documentation

In the official FIWARE Data Models Documentation[31] it is stated that it is recommended to reuse schema.org data types. According to the provided examples in the official documentation of Orion[6], the used data types are for example `String`, `Integer` and `Float`. It would be more convenient to use the proposed datatypes like `Number`, `Text`. New users and/or developers might struggle with this kind of inconsistency, therefore we recommend stick to the suggested ones. Especially when the specified format for the content has been decided to use JSON and the `Content-Type: application/json`. Now it is even more coherent to use the suggested JSON data types in all examples as suggested. Besides this, we think that providing a reference page with all the error codes and the error-handling, what to expect and how to deal with it, would be really helpful.

[6]`https://fiware-orion.readthedocs.io/en/master/user/walkthrough_apiv2/index.html`

3.2. Feature Requests

In this section we describe some high level feature requests which we think would improve Orion's functionality in a good way. This would increase Orion's usability but also help upcoming developers in the near future once these features have been taken into account.

Schema Validation

Nowadays every entity can be easily changed in data-types and adding or removing attributes, no matter who published this entity. There are some harmonized data models[31] provided by FIWARE, but not all applications have been covered yet. Hence, applications might expect different data-types or key-value pairs than they are actually going to receive since anyone could update the existing entity. Therefore, we are describing a feature request which can be realised in two different ways to prevent overwriting entities with other values:

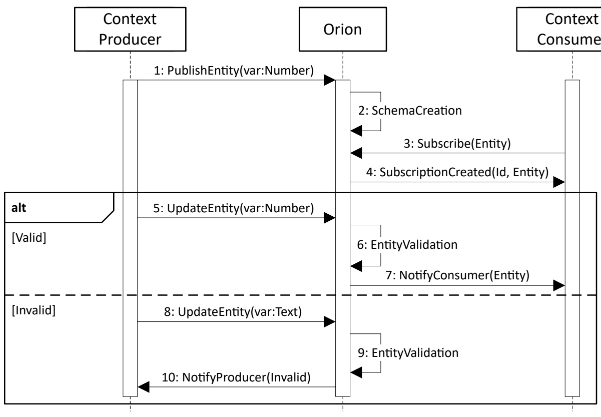
1. Centralized schema validation through Orion
2. Providing schemas for entity validation through consumers

Centralized schema validation through Orion

Our first described approach, the centralized entity validation, implements a data validation against a schema by Orion. This mechanism is depicted in the Figure 5.

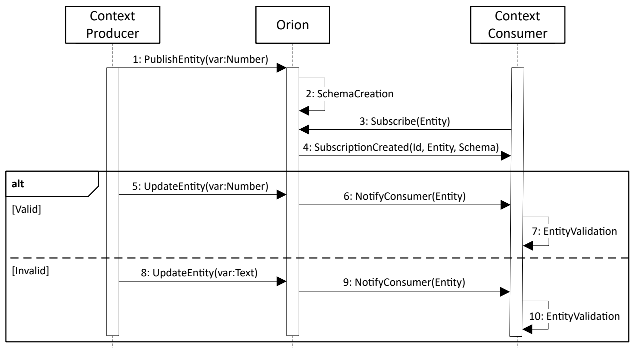
The context producer publishes its entity to Orion (Step 1). Based on this newly created entry, Orion could create the schema for this entity (Step 2). The consumer would subscribe to this entity as before (Step 3) and receives also the acknowledgment of the created subscription (Step 4). Now, when the context producer updates the entity (Step 5), Orion can check the received update of the entity (Step 6). Since the data is valid, all subscribers can be notified (Step 7). If the producer provides another format for the entity (Step 8), Orion would reject this update of an entity (Step 9) and notifies the producer about the wrong format (Step 10) instead of distributing the updated entity to the consumer.

We are aware that this feature of the data validation through Orion might slowdown the data-distribution but the subscribed consumers would benefit from it. Currently anyone can overwrite an entity where the subscribed consumers will receive then notifications. Another variety of this approach could be that the



Context Producer
Orion
Context Consumer
1: PublishEntity(var:Number)
2: SchemaCreation
3: Subscribe(Entity)
4: SubscriptionCreated(Id, Entity)
alt
[Valid]
5: UpdateEntity(var:Number)
6: EntityValidation
7: NotifyConsumer(Entity)
[Invalid]
8: UpdateEntity(var:Text)
9: EntityValidation
10: NotifyProducer(Invalid)

Figure 5. Sequence Diagram of a centralized schema validation through Orion



Context Producer
Orion
Context Consumer
1: PublishEntity(var:Number)
2: SchemaCreation
3: Subscribe(Entity)
4: SubscriptionCreated(Id, Entity, Schema)
alt
[Valid]
5: UpdateEntity(var:Number)
6: NotifyConsumer(Entity)
7: EntityValidation
[Invalid]
8: UpdateEntity(var:Text)
9: NotifyConsumer(Entity)
10: EntityValidation

Figure 6. Sequence Diagramm example of how Orion could provide schemas for a entity validation through context consumers

context producer provides next to the entity also the required schema so that Orion does not have to create its own schema for a new entity. Hence steps 1 and 2 would be becoming one. If the publisher does not provide the required schema, Orion could create one automatically based upon the received entity.

Providing schemas for entity validation through consumers
To avoid the slowdown of Orion, we are also proposing a second approach which is depicted in Figure 6. Based on the first published entity (Step 1), Orion would create an automatically JSON Schema (Step 2). After a context consumer subscribes to an entity (Step 3), Orion provides the created schema in the successful created subscription (Step 4). This schema could be used on the application side of the consumer to validate data, independent of whether the data are valid (Steps 5-7) or invalid (Steps 8-10). This approach has the advantage that Orion would not have to validate the data, only to notify the consumers about the update of an entity. Hence, the data-distribution would not lack of any performance and the delay, for forwarding the updated entity would be the same as before.

Providing a formal description through a schema also helps to provide a list of elements and attributes in a vocabulary and it also helps for documentation that is readable, as well for humans as for machines.

Communication
From our perspective, it would be really helpful to offer a new binding, like a WebSocket-Port. Nowadays, it is not easily possible for an HTML5 application to connect to a server which is not the deployment server due to the CORS-problem[32]. In addition to that, usually HTML applications are not offering a HTTP endpoint to receive the updates of an entity which is published by Orion to a HTTP REST endpoint. To connect a HTML based application it is required to use a NGSI Proxy which is capable of receiving POST notifications from Orion and forward this notification through an `EventSource` connection. Our proposed solution is depicted in Figure 7, where Orion offers next to the common HTTP REST/NGSI-Port also a `WebSocket`-Port. Incoming `WebSocket` connections from a `UserClient Browser`, in fact by a browser using `JavaScript`, could be handled by forwarding the data to the NGSI-Endpoint inside the HTTP-Port for a transparent way of using Orion's API. Sending data from the Orion to the HTML5-application could be easily done by sending the data through the `WebSocket` connection.

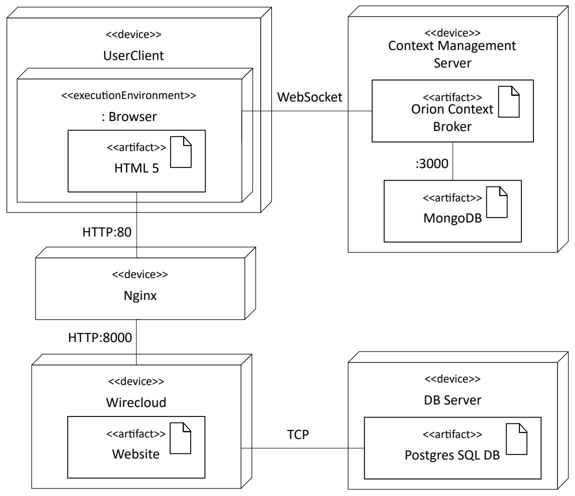


Figure 7. FIWARE Orion offering a WebSocket-Port to avoid the CORS problem

Of course, a `WebSocket` connection is a bit more complicated to handle than a HTTP, but this depends on the use case of the application. A `WebSocket` approach could help improve the communication for applications with a high frequency data rate. Instead of establishing a HTTP connection for every data exchange it might be easier and more efficient just by keeping an open connection[33].

4. IMPROVING FIWARE's Security

In this section we will discuss all issues that are related to security, which are ranging from mere feature request to severe vulnerabilities.

4.1. Authorizing Sensors

The southbound interface used by sensors is particularly sensitive. Sensors need to be authenticated, since - depending on the use case - it is important to trust their identity and their values. However, sensors are mounted out in the field, potentially anyone can get access to them and revealing their credentials is possible. By design, the potential damage that could be caused by a sensor that was tampered with should be minimized. Currently the IdM treats sensor accounts differently from other user accounts. This is generally a good idea, since, whenever a new user is created, the account needs to be manually activated by clicking an activation link that gets sent via email. This is infeasible for sensors. Unfortunately, there is also no possibility to define permissions for device accounts. Consequently, FIWARE only supports authentication but no authorization for sensors. When we look at our smart living example again and assume that a tenant of an apartment managed to get access to a temperature sensor's credential, these credentials could be used to mimic any other sensor, since any API-key, device-id and sensor data could be used without restrictions. Thus, these credentials could for example be used to trigger a fire alarm in a different building.

This example should make clear, that authorization is required here as well, which would limit misusing a sensor to only one case, which is sending fake values for the tampered sensor itself. Implementing this requirement would need a tighter integration of IDAS and IdM and some minor changes to the protocol. Currently, when a sensor submits data to IDAS, API-key and sensor-id are sent in the query string of the request[34]. However, the query string is entirely ignored by the PEP-Proxy and by XACML. Thus, these two parameters need to be part of the URL in order to become part of a permission rule. By requiring the device id to be identical to the sensor's account name and some similar constraint for the API-key, the IdM could easily authorize such requests without a lot of additional configuration or even the need of explicit permissions.

4.2. Multi Tenancy

As already briefly pointed out in section 2, FIWARE supports multi tenancy via the `Fiware-Service` HTTP header that needs to be part of every request. For every tenant the context broker creates a separate database with the name `orion-<FIWARE-Service>`. This ensures that data belonging to different tenants or applications is strictly separated.

Security-wise this behavior currently seems to be the most severe issue, since this header field – like all headers in general – is ignored by FIWARE's security infrastructure. Thus, if you are having a valid FIWARE account along with a set of permissions these permissions allow you to perform the same set of actions on any other tenant's data as well, since the value of the `Fiware-Service` header can be selected without any constraints. If this header does not match any of the existing databases, simply a new one is created.

To fix this issue, the PEP-Proxy needs to consider the value of this header field. The authorization could either be done by the PEP itself (e.g. by matching the header to some value provided by the IdM) or by the IdM, which requires the PEP to send the header field with its authorization request. Since the IdM allows to assign users to organizations, one solution could be to enforce the `Fiware-Service` header being identical to the name of a user's organization.

5. Improving Comet

Comet is the component that provides unified RESTful access to the historic context data (time series). Before going into details of the shortcomings of this component it needs to be stated that there are currently two initiatives to replace Comet and Cygnus – which are needed to create and to read time series data – in the future. These initiatives are called Draco[7] and Quantum Leap[8]. Since we haven't used any of these components yet, we cannot provide evaluation results either, but can refer to the documentation of these new FIWARE parts. Draco merely seems to be a re-engineered version of Cygnus that uses Apache NIFI[9] instead of Apache Flume. Quantum Leap, however, introduces a whole new technology platform with lots of new possibilities and is most likely suited to eliminate most of the problems we came across

[7] `https://fiware-draco.readthedocs.io`
[8] `https://smartsdk.github.io/ngsi-timeseries-api`
[9] `https://nifi.apache.org`

when working with Cygnus/Comet. It is basically an NGSI interface to common time series database like InfluxDB[10], RethinkDB[11] and Crate[12], with currently a clear emphasis on the later one. It subscribes for change notifications with the context broker, transforms the incoming messages and stores them in a time series database. Consequently, time series data can be visualized using popular open source solutions like Grafana[13]. Both of these new components are in an early stage and therefore still limited in their capabilities. When getting back to Comet, we need to take a closer look at the basic API functionality in order to understand its current shortcomings. Every request to retrieve data from comet has to include the type and id of the entity and the name of the desired attribute. Thus, the response contains a series of values for one attribute. This already turned out to be problematic in one use case, where we had to visualize values of mobile sensors, since getting the location of a sample requires a second request to Comet. Every request also requires a query parameter that limits the size of the response. Options are:

- **lastN:** limits the result to the N most recent samples. This value cannot exceed the configured maximum result length, which is originally 100.
- **hLimit/hOffset:** returns at most hLimit samples starting at hOffset, which allows for paginated results
- **dateFrom/dateTo:** allows to restrict results to a data range. This option has to be used with one of the previous restrictions

There is also a way to query aggregated results. In order to use this feature, a special Apache flume connector (NGSISTHSink[35]) needs to be configured, that stores *min*, *max* and *sum* values for all attributes and predefined periods in separate MongoDB collections. If this data sink is not used and therefore the collections of aggregated results are not present, Comet will NOT produce any error message, but returns empty results instead. Analyses showed that the resulting collections for aggregated information have at least the size of the actual time series data, although it just represents data that could be easily computed during a query. In fact, we ran into the following problem: In one of our projects we are running a sensor network, where every sensor produces a new sample every 30 seconds, resulting in 2880 data points per day. We also wanted to visualize these points over a user-defined period of time. Our

[10]https://www.influxdata.com/
[11]https://www.rethinkdb.com/
[12]https://crate.io/
[13]https://grafana.com/

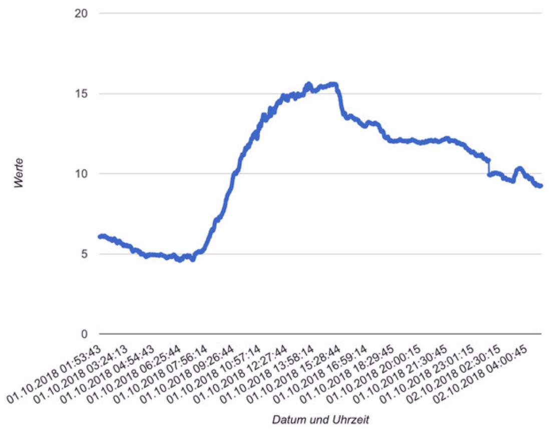


Figure 8. Temperature time series visualization using Comet (Screenshot from Wirecloud)

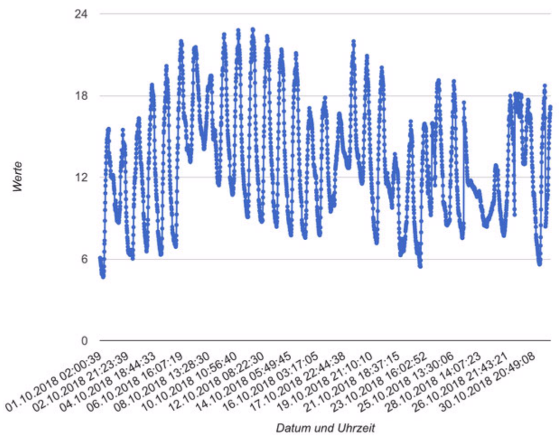


Figure 9. Using the refactored version of Comet (Screenshot from Wirecloud)

first result looked like shown in Figure 8, which should actually display all values for October 2018.

The *hLimit* parameter (and the maximum length configuration parameter) were set to 3000, consequently the data series stops after 3000 points, which covers slightly more than one day. To cover the whole month we would need a limit of about 90000 points, which does not make sense for a graphic that is probably 1500 pixels in width. What we actually wanted was a behaviour that allows us to cover all points for the given time period but automatically scales down values to a maximum of 3000 (or any other given value) points. This is a feature that is not supported by Comet, so we ended up in cloning the Comet repository and introducing the necessary changes ourselves. The result can be seen in Figure 9, which shows temperature in October 2018 reduced to 3000 points using the average of neighboring values.

We also implemented the functionality that was

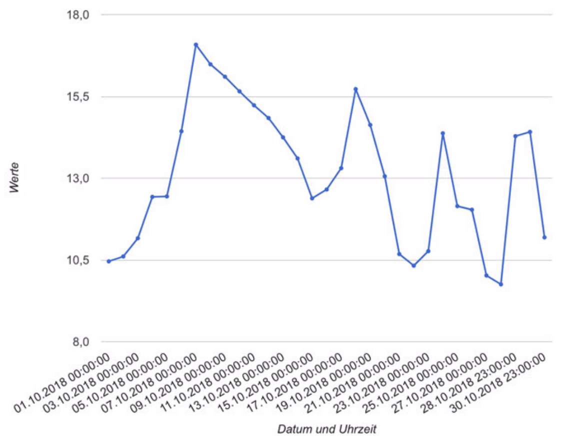


Figure 10. Daily maximum temperature (Screenshot from Wirecloud)

previously provided by the NGSISTHSink, however, without the need of any additional collections. Figure 10 shows the daily maximum temperature for October 2018.

6. CONCLUSIONS

FIWARE's platform is offering a wide range of tools to create applications for (smart) cities. One of the strengths of the FIWARE platform - apart from being open source - is that its components are well integrated and are working together with little configuration effort. Since all of them also come as docker images its is relatively straight forward to set up a basic environment that can be used for example as an IoT platform. Generally, using these tools should make life easier for application developers, integrators and end users. On the other hand, as we have figured out based on our experiments and code analysis, the technological and functional basis of most of the components we have investigated so far, is far from perfect, which is also partially reflected by FIWARE's internal quality metrics[36].

Although FIWARE's context broker Orion is considered to be one of the most mature and stable elements within the platform, we have identified several improvements when it comes down to the usage of the HTTP header fields, like providing a more standardized support for versioning. Another improvement could be achieved by using an opaque string, like the `ETag`, in the HTTP response header for a simple caching mechanism. These two suggestions will greatly improve the overall usability and performance without the need of an in-depth re-factoring. Another useful feature for Orion would be a schema validation or the additional WebSocket-Port. When it comes to Comet - the component used to access time series data - we have pointed out some missing functionality, like a missing error messages in case of an unused data sink. However, we have also revealed some missing functionality within FIWARE's security stack, like the authorization of sensors. This could be easily fixed without breaking any of the existing protocols. We are also convinced that users of the FIWARE platform will relatively quickly come across the same issues. On the other side, we see that currently a lot of development is going on. Nevertheless, we are strictly convinced that before new features and new components are about to be incorporated into the FIWARE platform, the emphasis should clearly be put on fixing issues and re-factoring of the existing code base - like it was already done with the IdM - in order to meet the need for a reliable, performant, modular and open source IoT and smart application infrastructure.
Our future work will involve monitoring of our here presented improvements. In addition, we are also planning to have a closer look at the non-functional requirements like stability, response times and performance based on automated testing. Performance can be considered from different points, like from the application side or from identifying possible bottlenecks inside the source code. This could also include verifying the readability of the source code.

Acknowledgment

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 767642 (L4MS).

References

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

[2] ——, "FI-GLOBAL: Building and supporting a global open community of FIWARE innovators and users," EU, Brussels, Belgium, Tech. Rep., 2016.

[3] ——, "A FIWARE-based SDK for developing Smart Applications," EU, Brussels, Belgium, Tech. Rep., 2017.

[4] ——, "Bringing FIWARE to the NEXT step," EU, Brussels, Belgium, Tech. Rep., 2017.

[5] FIWARE Team. (2011) What is FIWARE? [Online]. Available: https://www.fiware.org/2011/05/17/what-is-fiware/

[6] FIWARE Foundation. (2018) FIWARE Foundation. [Online]. Available: https://www.fiware.org/foundation/

[7] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, 2000, aAI9980887.

[8] FIWARE Orion Team. (2018) FIWARE-ORION Documentation. [Online]. Available: https://fiware-orion.readthedocs.io/en/master/index.html

[9] Open Mobile Alliance. (2018) NGSI Context Management. [Online]. Available: {http://www.openmobilealliance.org/release/NGSI/V1_0-20120529-A/OMA-TS-NGSI_Context_Management-V1_0-20120529-A.pdf}

[10] FIWARE IDAS Team. (2016) FIWARE IDAS Backend Device Management. [Online]. Available: https://catalogue-server.fiware.org/enablers/backend-device-management-idas

[11] D. Hardt. (2012) The OAuth 2.0 Authorization Framework. RFC 6749. [Online]. Available: https://tools.ietf.org/html/rfc6749

[12] Álvaro Alonso, A. P. Huertas, and J. Fox. (2018) Identity Management – KeyRock. [Online]. Available: https://fiware-idm.readthedocs.io/en/7.0.0/

[13] Álvaro Alonso and J. Fox. (2018) Identity Management – KeyRock. [Online]. Available: https://fiware-pep-proxy.readthedocs.io/en/7.5.1/

[14] Cyril Dangerville and Jason Fox. (2018) FIWARE Authzforce CE. [Online]. Available: https://authzforce-ce-fiware.readthedocs.io/en/latest/

[15] E. R. (Ed.). (2013) eXtensible Access Control Markup Language (XACML) Version 3.0. [Online]. Available: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html

[16] Álvaro Arranz and Jason Fox. (2018) FIWARE Wirecloud. [Online]. Available: https://wirecloud.readthedocs.io/en/stable/

[17] R. T. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing," RFC 7230, Jun. 2014. [Online]. Available: https://tools.ietf.org/html/rfc7230

[18] M. Masse, *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media, Inc., 2011. [Online]. Available: https://books.google.ch/books?id=eABpzyTcJNIC

[19] C. Rodríguez, M. Baez, F. Daniel, F. Casati, J. C. Trabucco, L. Canali, and G. Percannella, "REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices," in *Web Engineering - 16th International Conference, ICWE 2016, Proceedings*, ser. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 9671. Germany: Springer Verlag, 2016, pp. 21–39.

[20] F. Palma, J. Dubois, N. Moha, and Y.-G. Guéhéneuc, "Detection of REST Patterns and Antipatterns: A Heuristics-Based Approach," in *Service-Oriented Computing*, X. Franch, A. K. Ghose, G. A. Lewis, and S. Bhiri, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 230–244.

[21] F. Palma, J. Gonzalez-Huerta, N. Moha, Y.-G. Guéhéneuc, and G. Tremblay, "Are RESTful APIs Well-Designed? Detection of their Linguistic (Anti)Patterns," in *Service-Oriented Computing*, A. Barros, D. Grigori, N. C. Narendra, and H. K. Dam, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 171–187.

[22] S. Vinoski, "RESTful Web Services Development Checklist," *IEEE Internet Computing*, vol. 12, no. 6, pp. 96–95, Nov 2008.

[23] M. Stowe, *Undisturbed REST: A guide to designing the perfect API*. Mulesoft, 2015. [Online]. Available: https://books.google.de/books?id=Gg0sCgAAQBAJ

[24] L. Richardson and S. Ruby, *RESTful web services*. O'Reilly Media, Inc., 2008.

[25] FIWARE Orion Team. (2018) Fiware-Orion Code. [Online]. Available: https://github.com/telefonicaid/fiware-orion/blob/d6f308616795ed6cdaad86f739fd19717157def3/src/lib/rest/rest.cpp#L1548

[26] R. T. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content," RFC 7231, Jun. 2014. [Online]. Available: https://rfc-editor.org/rfc/rfc7231.txt

[27] ——, "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests," RFC 7232, Jun. 2014. [Online]. Available: https://rfc-editor.org/rfc/rfc7232.txt

[28] H. F. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," RFC 2616, Jun. 1999. [Online]. Available: https://rfc-editor.org/rfc/rfc2616.txt

[29] A. Neumann, N. Laranjeiro, and J. Bernardino, "An analysis of public rest web service apis," *IEEE Transactions on Services Computing*, pp. 1–1, 2018.

[30] G. Levin. (2016) Restful apiversioning insights. [Online]. Available: https://dzone.com/articles/restful-api-versioning-insights-1

[31] FIWARE Foundation. (2016) FIWARE-DATAMODELS - Data Models Guidelines. [Online]. Available: https://fiware-datamodels.readthedocs.io/en/latest/guidelines/index.html/

[32] Mozilla Developer Team. (2018) MDN Web Docs - Cross-Origin Resource Sharing. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

[33] V. Pimentel and B. G. Nickerson, "Communicating and displaying real-time data with websocket," *IEEE Internet Computing*, vol. 16, no. 4, pp. 45–53, July 2012.

[34] FIWARE Team. (2019) FIWARE-IoT-Stack Device API. [Online]. Available: https://thinking-cities.readthedocs.io/en/master/device_api/index.html

[35] F. G. Márquez. (2018) NGSISTHSINK. [Online]. Available: https://fiware-cygnus.readthedocs.io/en/1.3.0/cygnus-ngsi/flume_extensions_catalogue/ngsi_sth_sink/index.html

[36] J. J. Hierro. (2018) FIWARE GE QA labels. [Online]. Available: https://docs.google.com/spreadsheets/d/1lXXp-BU14xAoB4b2OTbsdtWhmmEQStneZuJadRk_H6o/edit#gid=1326252107