Developing water monitoring solutions with FIWARE defining, creating, populating, and extending digital twins

Gareth Lewis[1], Brett Snider[1], Lydia Vamvakeridou-Lyroudia[1,2], Albert S Chen[1], Slobodan Djordjević[1] and Dragan A Savić[1,2]

[1] Centre for Water Systems, University of Exeter, Harrison Building, North Park Road, Exeter, EX4 4QF, UK
[2] KWR Water Research Institute, Groningenhaven 7, P.O. Box 1072, 3430 BB Nieuwegein, the Netherlands

g.lewis2@exeter.ac.uk

Abstract. The water industry is often and unfairly criticised for being relatively slow to adopt IT as part of its operating processes. To address this, the FIWARE and Fiware4Water projects were tasked with developing open-source data definition, management, and processing, and applying it to the water sector. In this work, we are reviewing our experiences with developing FIWARE-based solutions for the aqua3s[1] and Fiware4Water[2] projects across the whole project lifecycle, from our initial of installing and learning to work with FIWARE context brokers[3], through building and maintaining synthetic digital twins for load and edge-case testing development and debugging and finally looking at longer-term data management and archiving. Through our work on these projects, we have started to develop a wealth of best-practice, strategic choices, and approaches to be avoided for successful and long-term context broker usage.

1 Introduction

A digital twin is usually defined as some form of virtual representation that serves as the real-time digital counterpart of a physical object or process. For the Fiware4Water project [2], the Centre for Water Systems partnered with South West Water on a 'smart metering and citizen engagement' project, to develop a digital twin representation of a smart metering network installed in Devon, UK. For the aqua3s project [1], multiple digital twins were constructed to model water delivery and monitoring networks across multiple European pilots.

For both projects, the digital twin goals were similar, in that both projects required real-world data to be defined, modelled, and collected for digital twin representations. Once collected, data could be stored long-term and analysed and processed in manners that were suitable for the projects.

From a technical architecture perspective, these projects can be considered through the model-view-controller paradigm, in that data is collected in the model perspective, processed in the controller, and presented to users through the view. Traditionally, the model perspective would be realised through a relational database, either an SQL or no-SQL. However, for the smart metering and water sensing projects, this approach isn't ideal given the scope for dissimilar data formats, e.g., different types of sensing devices recording different types and formats of data, and the limited need for 'heavy' database functionality, e.g., relationships between different tables.

2 FIWARE as a technology for digital twins

FIWARE [4] describes itself as a curated framework of open-source platform components, based around several NGSI specifications [5] and a set of smart data models [6], with a context broker forming the core of a digital twin architecture.

Figure 1 details a typical FIWARE-based digital twin and highlights the three key context processes. Initially, contexts are generated with an IoT agent that collects data from devices, e.g., sensors, smart meters etc, and stores it on the context broker. Context transformation describes processes that use existing contexts to create new contexts or modify existing contexts, whilst context consumption describes processes that use existing contexts as inputs to present information to users.

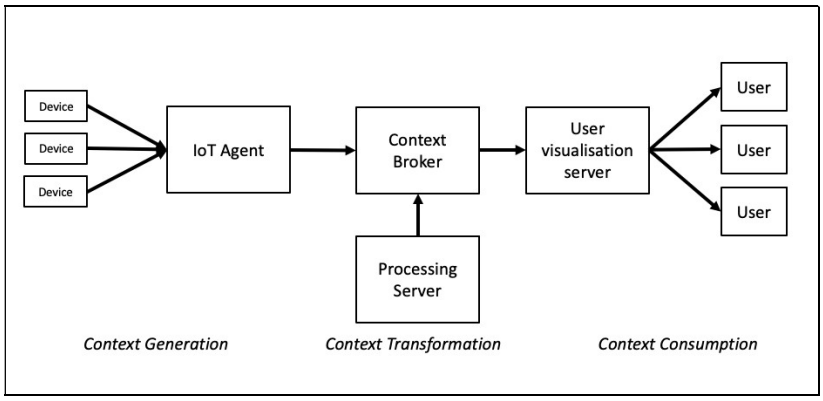


Figure 1. Typical FIWARE broker installation.

2.1 FIWARE for aqua3s

For the aqua3s project, a digital twin was developed through the periodic collection of sensor data in water distribution networks. The digital twin's users were SCADA staff who could use processed context data to assess the state of the water network and support decision making for operational issues. As per the FIWARE context broker architecture, sensor data was collected through the IoT Agent to generate appropriate context data using a pre-existing 'device' smart data model [7].

These device contexts were used as inputs to an alert process which created alert state contexts which were then used to determine the state of the system. Both the device and alert context were then consumed into user visualisations, Figure 2, with the left-hand image showing sensors visualised in a geographic context with alert context used to colour-code the sensors (green showing no current alert) along with related quantitative data. The right-hand image showing historic quantitative data from the sensor contexts.

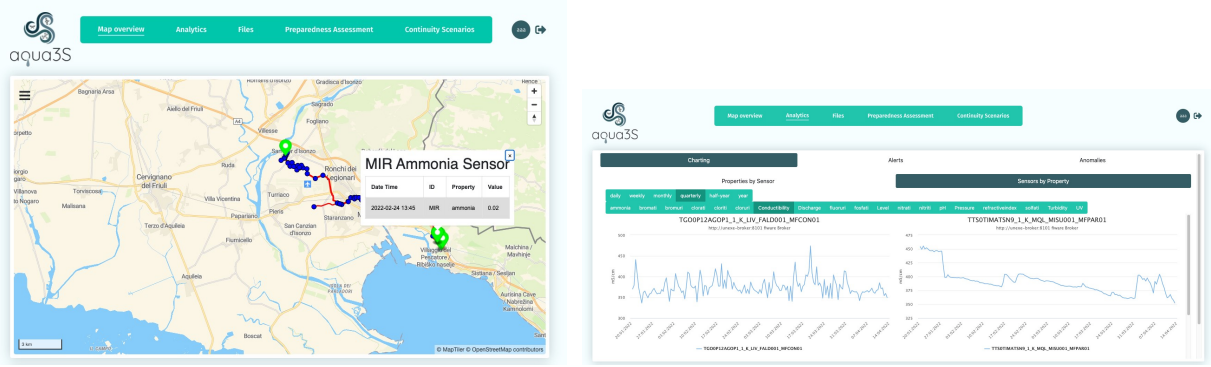


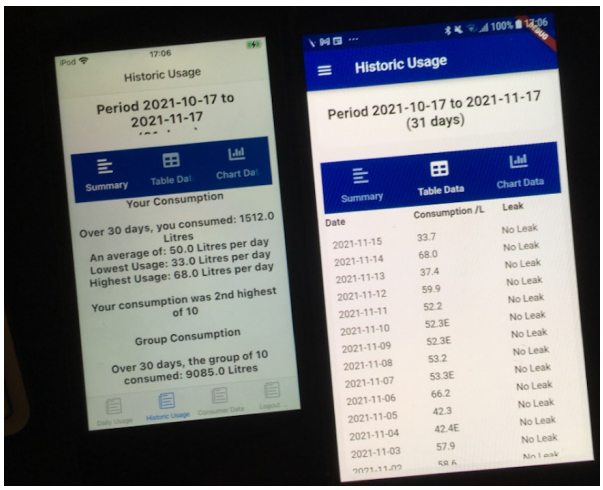
Figure 2. aqua3s user visualisations from FIWARE data. Current sensor data visualised with maplibre [8] (left), historic sensor data visualised with highcharts.js [9] (right).

2.2 FIWARE for Fiware4Water

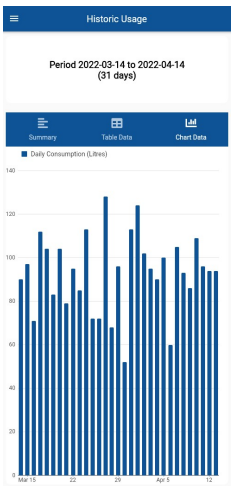
For Fiware4Water, a digital twin was developed through the daily collection of customer water consumption through smart meters. The digital twin had two distinct users: staff at South West Water who would use the data to determine customer-side leaks through a leak reporting service and customers who could monitor their personal consumption through a mobile app. As per the FIWARE context broker architecture, smart meter data was collected daily and then processed to give daily individual and group consumption figures. This data was then accessed through the two visualisation applications on an ad-hoc basis, ***Figure 3***. The top image shows the staff leak reporting application, ranking meters by number of recent leak alarms and showing recent historic customer consumption. The bottom image shows the customer app, showing daily and historic consumption in a textual form, along with customer consumption ranking in their local group and a longer-term graphical consumption graph.

For this project, a 'WaterConsumptionObserved' model was developed and made available to the smart data models repository [10].

(a)



(b)



(c)

Figure 3. Fiware4Water user visualisation from FIWARE data. Leak reporting application (a), customer consumption application (b)-(c).

3 Results

3.1 What worked well?

In general, FIWARE has been a good solution for both projects. FIWARE has a general model of 'ease of use' which comes across in most aspects of its operations. Providing context brokers as Docker images [11], [12] makes the installation and operation of brokers straightforward, notwithstanding docker issues. The context broker http request interface effectively contains just 4 commands (POST, PATCH, GET and DELETE) and the ngsi-ld/v1 spec demonstrates these operations clearly, far more so than the broker documentation, which can be hard to find.

The key benefit of FIWARE has been the use of contexts and smart models. Moving away from SQL and explicit table definitions to the JSON-LD format allows smart models to be authored in an extremely flexible manner using key-value pairs and is particularly useful in situations where a device may have multiple properties that it records. This was a huge benefit for both research projects as it enabled researchers to spend their time solving important research problems and not determining the most suitable table specification to capture arbitrary data. It should be noted that moving away from direct SQL interaction is something of a trend in web development, with Django's object relationship mapping [13] and SQLAlchemy [14].

3.2 What didn't work so well?

The first challenge of working with FIWARE is determining which broker to use as there is a wide range of solutions available which all appear to offer differing levels of functionality / adherence to the ngsi-ld specification, organisation, and requirements. For Fiware4Water, Stellio [15] was used which was presented as a single component broker which worked well for the project but did not support multiple-tenant (a core requirement for aqua3s) and tended to use a lot of resources, making it difficult to run on lightweight virtual servers for development and testing. Conversely, aqua3s used Orion-LD [12] for 'short-term' data storage and Cygnus [16] for longer-term storage, though approach was problematic to make work reliably and led to significant simplification of the device model that was used.

As expected, brokers had performance limitations, concerning sequential http requests, with a typical use case being the retrieval of entities. Whilst the ngsi-ld/v1 supports the entities command, it is limited to the number of entities it will return to keep the size of data packets down, so the normal approach is to query the broker for the number of instances of a given entity and then retrieve entities individually. This can be addressed to some degree with threading requests, but it does demonstrate the nature of http requests and the need to develop algorithms that are sympathetic to data collection.

One approach to mitigate this is the use of FIWARE's notification model, in that the broker can be programmed to send a request to a remote device upon the receipt of a particular request itself. The use case here is that when the IoT Agent sends a PATCH request to the context broker, the context broker can then send a notification to another http request server to perform some function based on the entity that has been patched. This was problematic for both the aqua3s and Fiware4Water projects, in that for aqua3s Orion-LD notifications appeared to be unreliable and for aqua3s, the operation Stellio broker could not resolve to local development machines for testing. Whilst this would clearly work in an operation context, we were reluctant to put untested functions straight into an operational environment.

From our experiences on both projects, it became apparent that broker functionality was geared around operations rather than development / testing activities. This became particularly apparent when brokers were loaded with synthetic data for testing. Typically, this would involve creating a new broker and a set of contexts using the POST request for each instance of context data. For Fiware4Water, we had a population of 100 smart meters with daily collections, giving a yearly backlog of around 36,500 data points. Aqua3s presented more of an issue with a pilot consisting of up to 20 devices and device data collected every 15 mins, giving around 100 data points per day per device, so 730,000 contexts as the total backlog to post, which was unrealistic to upload to the broker.

4 Developing a bespoke brokerage

During the development of the aqua3s project, development of context transformation and consumption functionality was delayed due to issues with setting up Cygnus to reliably manage historic data. Initially, delays were met with creating lightweight historic broker substitutes, but as the delays continued, it became apparent that little extra work was required to convert the 'lightweight historic broker' into a generic context broker. This work wasn't undertaken lightly as we would have preferred to use Stellio as an off-the-shelf solution, given that it was already being used, and working, on the Fiware4Water project. However, Stellio's resource requirements made it unusable on the lightweight development server we were running and Stellio's lack of multi-tenant support coupled with our lack of understanding of multiple container Dockerisation made it a non-starter.

The bespoke brokerage was built in Python using Flask as the http request frontend and SQLite as a backend database. Protocols were implemented using a mixture of the ETSI standards [5] and black box testing on Stellio by sending commands to the broker and comparing the Stellio results with the bespoke results. The broker was not designed to be anything like a full implementation, just an implementation of the features that were being used for development.

4.1 Benefits of the bespoke brokerage

The most obvious benefit of the bespoke brokerage was the development of an alternate patching strategy for building testing backlogs. An additional request was implemented that took an array of patch data for a given entity, rather than relying on multiple PATCH requests for each data point. This made it possible to build large backlogs in realistic times. For example, creating a broker with 3 tenants, each with 5 devices and 3 properties per device and 10 days of data with a 15-minute data point interval takes around 270 seconds on a broker using a 'normal' patching approach, of one patch per update, or 28 seconds using a patch array containing all the patch data in one request. Whilst this is clearly not a normal operational activity for a context broker, it's a key development / testing activity.

Having complete control over the broker opened a lot of opportunities based around status and logging. Working across multiple locations and time zones, it was always difficult to know what the status of a shared broker was, in terms of what tenants had been set up and what entities were in those tenants. Adding an unexe-broker/v1/version command returned the current state of the broker across all tenants, detailing all entity types and their instances.

Another area for improvement was broker logging. Whilst Stellio provides a lot of logging information through the Docker console, all the useful request and response information tends to get lost amongst internal component messages, typically Kafka and PostgreSQL. Whilst this is useful for Stellio developers, it didn't help our work. We realised that the logging interface could then be extended into a transaction log, which could prove invaluable for recording and re-running problematic broker requests, particularly during client testing sessions. This could also be coupled with an archival approach such that a copy of the broker can be dumped to disk prior to testing and any issues that occur would be stored in a transaction log. After testing, developers could then reload the initial archive and re-run the transaction log to determine the cause of the issue.

4.2 Issues of the bespoke brokerage

The most pressing issue of the bespoke brokerage is that it not FIWARE compliant and it doesn't completely follow the ngsi-ld specs. It follows the specs to the degree that we are using the specs in our projects, but it would be completely unrealistic to drop the broker into another project and expect it to work. However, the broker was not designed for such open-ended use cases, so it's not too much of a concern.

There is also the possibility that applications developed with the bespoke brokerage could 'drift' from FIWARE/ngsi specifications and end up using requests that are not supported by FIWARE brokers. This is a real risk, in that FIWARE brokers tend to implement interpretations of the ngsi specification and broker updates have included changes to request arguments. This is the nature of

working with off-the-shelf software that is in development. From our side, we have developed a suite of black box tests for working with brokers, so issues can be raised and, hopefully, addressed.

The broker was developed as a Python/SQLite [17] application, so developer-time efficiency trumped run-time efficiency, therefore, the broker is likely to run into performance issues sooner or later. These are likely to concern the broker's ability to process multiple simultaneous clients, process database calls and so on. Again, the broker wasn't designed for such open-ended use cases, and it currently meets its performance goals.

5 Conclusions

The fundamental question to ask at this point is 'does it make sense to use FIWARE?' and, of course, the answer is 'yes, absolutely'. Like all technologies and frameworks, even the good ones are a bit rough around the edges and the key thing with them is that their value massively outweighs their issues. In general, the LD-JSON based context broker is a huge win for developers as it enables them to spend their time concentrating on solving real-world problems and not re-solving SQL representation issues. Additionally, ETSI's ngsi-ld/v1 specification makes a lot of sense for the use cases that developers will generally encounter with broker development.

For us, the bespoke brokerage has worked very well to support our development work on both projects. For aqua3s, it was a useful solution to continue with meaningful development and testing work whilst the 'proper' solution was in development. For the Fiware4Water project, although we had a working operations broker from day one of the project, the bespoke brokerage gave us a shared broker to use for development and testing work given our reluctance to mix development and operations environments.

The broker has allowed us to create development and test functionality that may be of little to no use to the operational role of FIWARE brokers but has made a huge impact on the work we have undertaken.

Developing a broker has given us a lot of insight into how brokers can work and the relationship between smart data models and underlying broker functionality. At the start of these projects, there was a conventional wisdom that smart data models should be small, in much the same way that classes are 'small', or at least of a single responsibility. The use of 'observedAt' in smart models effectively breaks a single model up into parts, given that the observedAt element of the model is stored in a time-based manner. From that, it makes sense to create fewer smart models with more observedAt elements as the model can be retrieved with a single http request. However, this may be problematic with Cygnus.

This paper isn't a stick to hit FIWARE with, it's just a report on the issues we ran into with doing development and testing work with FIWARE and what we did to address them. As researchers, we can 'travel light and break things' with our research and highlight findings that we think may be of interest to the broader development community.

Acknowledgements

The work presented in this paper was funded by the ongoing EC H2020 aqua3S (GA 832876), Fiware4Water (GA 821036) and LOTUS (GA 820881) projects.

References

[1] Aqua3s project homepage https://aqua3s.eu/ (accessed Jul. 12, 2021)
[2] Fware4Water homepage https://www.fiware4water.eu/ (accessed Jul. 12, 2021)
[3] FIWARE Components https://www.fiware.org/developers/catalogue/ (accessed Apr. 01, 2022)
[4] Fiware https://www.fiware.org/ (accessed Jul. 12, 2021)
[5] ETSI Context Information Management (CIM); NGSI-LD API
[6] FIWARE Smart Data Models https://www.fiware.org/smart-data-models/ (accessed Apr. 01, 2022)
[7] FIWARE DataModel.Device https://github.com/smart-data-

models/dataModel.Device/blob/master/Device/README.md (accessed Apr. 01, 2022).
[8] MapLibre Homepage https://maplibre.org/ (accessed Apr. 01, 2022)
[9] Highcharts Homepage https://www.highcharts.com/ (accessed Apr. 01, 2022)
[10] FIWARE DataModel.WaterConsumption https://github.com/smart-data-models/dataModel.WaterConsumption/tree/master (accessed Apr. 01, 2022)
[11] Stellio Docker Image https://github.com/stellio-hub/stellio-context-broker (accessed Apr. 01, 2022)
[12] FIWARE Orion-LD Docker Image https://github.com/FIWARE/context.Orion-LD (accessed Apr. 01, 2022)
[13] An introduction to the Django ORM https://opensource.com/article/17/11/django-orm (accessed Apr. 01, 2022)
[14] The Python SQL Toolkit and Object Relational Mapper
[15] Stellio Context Broker https://github.com/stellio-hub/stellio-context-broker/blob/develop/API_Quick_Start.md (accessed Jul. 12, 2021)
[16] Connecting Orion Context Broker And Cygnus https://fiware-cygnus.readthedocs.io/en/0.13.0/user_and_programmer_guide/connecting_orion/index.html (accessed Apr. 01, 2022)
[17] SQLite Homepage https://www.sqlite.org/index.html (accessed Apr. 01, 2022).