



Grant Agreement No.: 723174

Call: H2020-ICT-2016-2017

Topic: ICT-38-2016 - MEXICO: Collaboration on ICT

Type of action: RIA



D3.4: SmartSDK Reference Models and Recipes v2

Revision: v.1.0

Work package	WP 3
Task	Task 3.3
Due date	28/02/2018
Submission date	07/03/2018
Deliverable lead	MARTEL
Version	1.0
Authors	Federico M. Facca (MARTEL), Tomas Aliaga (MARTEL)
Reviewers	Daniele Pizzolli (FBK)

Abstract	This deliverable summarizes the work conducted in SmartSDK in relation to reference architectures and Data Models for cloud native smart applications based on FIWARE Data Management and IoT Enablement Generic Enablers.
Keywords	Architecture Patterns, Data Models, FIWARE, Data Management, IoT Service Enablement

Document Revision History

Version	Date	Description of change	List of contributor(s)
V1.0	05/03/2018	Applying review changes	Nestor Velasco Bermeo (ITESM), Daniele Pizzolli (FBK), Tomas Aliaga (MARTEL)
V0.3	27/02/2018	Merging contributions and review	Tomas Aliaga (MARTEL),
V0.2	16/02/2018	Updated Content	Tomas Aliaga (MARTEL), Federico M. Facca (MARTEL)
V0.1	12/02/2018	Table of Content	Federico M. Facca (MARTEL)

Disclaimer

The information, documentation and figures available in this deliverable, is written by the SmartSDK (A FIWARE-based Software Development Kit for Smart Applications for the needs of Europe and Mexico) – project consortium under EC grant agreement 723174 and does not necessarily reflect the views of the European Commission. The European Commission is not liable for any use that may be made of the information contained herein.

Copyright notice

© 2016 - 2018 SmartSDK Consortium

Project co-funded by the European Commission in the H2020 Programme		
Nature of the deliverable:		R
Dissemination Level		
PU	Public, fully open, e.g. web	✓
CI	Classified, information as referred to in Commission Decision 2001/844/EC	
CO	Confidential to SmartSDK project and Commission Services	

* R: Document, report (excluding the periodic and final reports)

DEM: Demonstrator, pilot, prototype, plan designs

DEC: Websites, patents filing, press & media actions, videos, etc.

OTHER: Software, technical diagram, etc.

EXECUTIVE SUMMARY

SmartSDK provides a set of ready to use “recipes” to develop smart applications in the Smart City, Smart Healthcare, and Smart Security domains. Such recipes are based on: components (i.e. Generic Enablers and Specific Enablers), data models (i.e. NGSI formalisation of the data exchanged among components) and reference architectures (i.e. the combination of components and data models to support production grade requirements).

This deliverable documents the work done in the project to support the definition of recipes and data models (including tooling support) and implementations for some of them.

Regarding the recipes, SmartSDK starts by analysing both FIWARE Reference Architectures for Data/Context Management and IoT Service Enablement adding an analysis of how Cloud Architecture Patterns (such as High Availability and Scalability) can be applied to FIWARE Reference Architecture. Starting from the most relevant GEs used to develop smart applications in SmartSDK, the deliverable documents how their deployment architecture can be modified to support high availability and scalability. Starting from the above deployment architecture and the analysis of best tools to implement it, this document defines concrete recipes for Docker Compose that enables the deployment and management of such architectures in few clicks. Covered key FIWARE components include:

- API Security framework
- Data Management (Orion, Cygnus, QuantumLeap, STH)
- IoT services enablement (UL, LWM2M and JSON agents)

Concerning the data models, SmartSDK analyses the current FIWARE data models in light of SmartSDK application scenarios. For each application scenario developed in the project, we identified which existing FIWARE data models would be reused in SmartSDK. The re-usage of the data models will ensure the validation and revision of existing data models in the context of the EU-Mexico collaboration, and will allow SmartSDK to contribute to such data models with novel reference data sets. Starting from the reused data models, SmartSDK also defined new data models to satisfy the application scenario requirements. Newly developed data models include:

- Alert data model that supports scenarios in which a smart platform sends alerts related to traffic jams, accidents, weather conditions, high level of pollutants and so on.
- Building data model to support the modelling of building and activities occurring in those.
- PollenLevelObserved data model representing the current quantity and allergen level of pollens.
- Smart POI data model defines an interactive point which provides information, entertainment or co-creation tools to citizens.
- Smart Spot data model defines a set of resources related to a physical device and the technology to provide a Smart Point of Interaction.
- Transportation Schedule data model supports the route planning process based on public transports routes.
- User Context data model let developers describe the context of a given (anonymised) user, e.g. the activity he is currently performing or his current location.
- VideoObject data model allows the storage of recorded surveillance video metadata and to annotate them with related detected security events.
- VisualObject describes identified objects in a video streaming.
- Physical Test and Control Test data models collect information from patients’ sensors in the

context of Health.

TABLE OF CONTENTS

EXECUTIVE SUMMARY	3
TABLE OF CONTENTS	5
LIST OF FIGURES	6
LIST OF TABLES	7
ABBREVIATIONS	8
1 INTRODUCTION	9
1.1 Design Principles.....	10
1.2 Structure of the deliverable	11
1.3 Audience.....	11
2 SMARTSDK ARCHITECTURE PATTERNS.....	13
2.1 Overview FIWARE Data/Context Management and IoT Services Enablement architecture patterns 13	
2.1.1 Support for generation of FIWARE Architecture Diagrams.....	15
2.2 Overview of cloud architecture patterns	16
2.2.1 Basic concepts related to distributed systems	16
2.2.2 Scalability Pattern.....	18
2.2.3 High Availability Pattern.....	18
2.2.4 Multi-site pattern	19
2.2.5 Co-locate Pattern	19
2.2.6 Queue-Centric Workflow Pattern.....	19
2.3 Methodology	20
2.4 SmartSDK Architecture Patterns.....	20
2.4.1 Context Broker	21
2.4.2 Comet Short Term History (STH).....	22
2.4.3 Cygnus.....	23
2.4.4 IDAS.....	24
2.4.5 CKAN.....	25
2.4.6 QuantumLeap	26
2.4.7 API Umbrella	28
2.5 Realization of Architecture Patterns recipes in SmartSDK.....	29
2.5.1 Foundation.....	30
2.5.2 Documenting and Developing Architecture Patterns recipes.....	33
2.5.3 SmartSDK Architecture Pattern recipes.....	34
2.5.4 Roadmap for SmartSDK Recipes.....	37
3 SMARTSDK DATA MODELS.....	38
3.1 Overview of existing FIWARE data models.....	38
3.2 Documenting and Developing Data Models	40
3.2.1 Automated support to FIWARE Data Model validation.....	41
3.3 Novel SmartSDK Data Models	43
3.3.1 Smart City.....	43
3.3.2 Smart Security	44
3.3.3 Smart Health.....	44
4 CONCLUSIONS AND FUTURE WORK.....	45
REFERENCES.....	46

LIST OF FIGURES

Figure 1: SmartSDK's cookbook concept.....	9
Figure 2: SmartSDK Design Principles.	10
Figure 3: Data/Context Management and IoT Service Enablement Southbound-Northbound Architecture.....	14
Figure 4: Availability and Partition Tolerance.....	18
Figure 5: Consistency and Partition Tolerance.....	18
Figure 6: HA Deployment.	19
Figure 7: Multisite deployment.....	19
Figure 8: Queue-Centric Workflow.	20
Figure 9: High Available and Scalable Orion Context Broker Architecture Pattern.....	21
Figure 10: High Available and Scalable Comet STH Architecture Pattern.....	23
Figure 11: Cygnus Architecture Pattern.	23
Figure 12: High Available and Scalable Cygnus Architecture Pattern.....	24
Figure 13: IDAS Architecture Pattern.....	24
Figure 14: High Available and Scalable IDAS Architecture Pattern.	25
Figure 15: CKAN Architecture Pattern.	25
Figure 16: High Available and Scalable CKAN Architecture Pattern.	26
Figure 17: Simple deployment of QuantumLeap.....	27
Figure 18: HA and Scalable deployment of QuantumLeap.....	27
Figure 17: Simple deployment of API Umbrella w/o analytics.....	28
Figure 18: HA and Scalable deployment of API Umbrella w/o analytics.....	29
Figure 17: Docker Swarm.	31
Figure 20: Architecture of the MongoDB replicaset recipe.	34
Figure 21. Example of outcomes of FIWARE Data Model validator.	42

LIST OF TABLES

Table 1: FIWARE data models by Scenario40

ABBREVIATIONS

API	Application Programming Interface
DNS	Domain Name Server
GE	Generic Enabler
GEri	Generic Enabler reference implementation
HA	High Availability
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
IP	Internet Protocol
REST	REpresentational State Transfer
TCP	Transmission Control Protocol

1 INTRODUCTION

The creation of common principles is the base for any activity that aims to scale beyond its initial promoters. In the case of Smart applications, i.e. applications applying intelligent decisions on top of acquired data, such common principles can be translated to common components (i.e. APIs), common architectures, and common data models. FIWARE is now reaching maturity and its adoption is spanning beyond Europe. The ability to leverage on the work done in Europe in other context, such as Mexico, is crucial to ensure that models and architectures defined are enough general to cover the specific needs and regulations of different countries. This process, also often referred to as “de-facto” standardisation, is crucial to facilitate the “replication” of platform and application deployments based on FIWARE.

While FIWARE so far mostly focused on the APIs definition and their basic interaction, SmartSDK focused its efforts to support his wider adoption by:

- ➔ Developing of reference data models to be used in different smart scenarios, ensuring interoperability across different applications (tackling the same data) and facilitating the deployment of applications across different sites.
- ➔ Providing tool support for the creation of novel data models.
- ➔ Developing of reference architectures that support the production grade deployment of applications. Such architectures need to deal with important requirements, such as High Availability and Scalability.
- ➔ Providing tool support for the deployment of architecture patterns through modern container management solutions and creating visual diagrams for such architecture patterns.

As such, SmartSDK developed the first FIWARE’s “cookbook” for developing smart applications in the Smart City, Smart Healthcare, and Smart Security domains. The process to achieve such goals started from looking into applications developed so far within Europe and Mexico (using FIWARE or alternative Open Source technologies), analysing them and using them to create “recipes” and “ingredients” for developing applications in the Smart City, Smart Healthcare, and Smart Security domains.

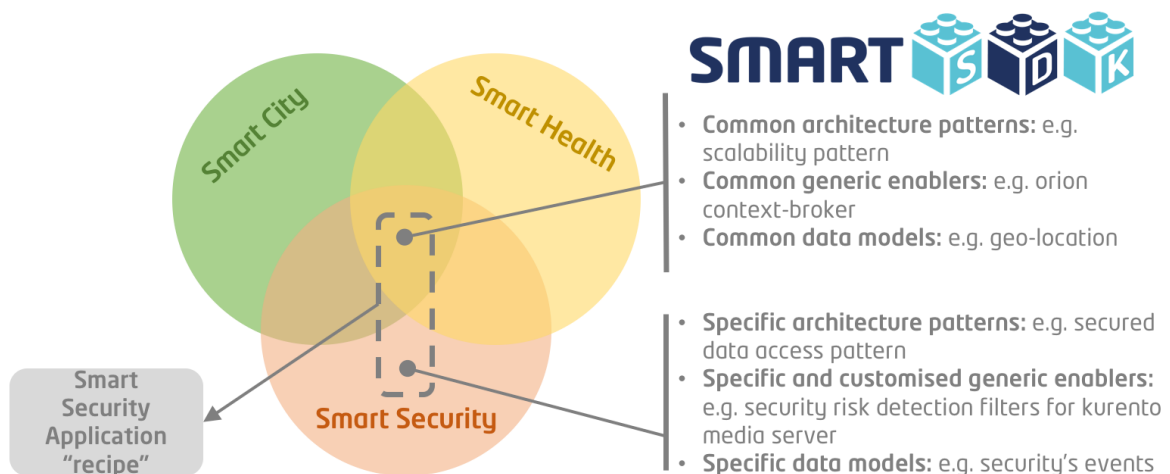


Figure 1: SmartSDK’s cookbook concept.

Such “cookbook”, as depicted in Figure 1, is based on: a set of architecture patterns (i.e. the basic cooking processes), a set of Generic Enablers (i.e. the basic ingredients) and a set of data models (i.e. the spices and flavours binding the ingredients through the cooking process).

While application scenario specific activities are carried out in the Application work package (WP2), this deliverable provides general reference architectures and data models for the development of smart

applications at scale, based on a set of design principles derived from FIWARE best practises. This deliverable includes, in addition to the list of achievements (for implementation details see the published repositories and artefacts at <https://smartsdk.eu/>), well documented guidelines for the development of new recipes and new data models, that may be employed by future project as reference for their activities in this context.

1.1 Design Principles

SmartSDK follows the design principles of FIWARE and brings it to its full potential, by moving from single enabler architectures to those which include multiple enablers. To this aim, SmartSDK focuses on the fundamental glue among enablers: data models and architectures.



Figure 2: SmartSDK Design Principles.

- ➔ **Restful APIs.** Restful APIs are programmable interfaces that exploit REST¹ (REpresentational State Transfer) architectural style to produce and consume data in a lightweight communication infrastructure. REST, which typically runs over HTTP (Hypertext Transfer Protocol), has several architectural characteristics: 1) Decouples consumers from producers; 2) Stateless existence; 3) Able to leverage a cache; 4) Leverages a layered system; 5) Leverages a uniform interface. All FIWARE Enablers offer Restful APIs, and SmartSDK, in contributing to FIWARE, will follow this design principle.
- ➔ **Reusability and Openness.** Software development activities often face problems already met by other developers before. Thus, the ability to capitalize on these developments is crucial to allow developers to focus on their core goals rather than having to build the foundation of their software before focusing on the core specific features of their applications. Reusability is at the core of FIWARE design: Generic Enablers are reusable software components, that developers may compose as Lego® bricks to build their application. Beyond that, through the evolution process from a European project to an Open Source community with its own governance, FIWARE embraced the Open Source principles to support the wider possible diffusion and impact of its achievements. Beyond the initial Open specifications, as of today all official Generic Enabler reference implementations are available as Open Source (mostly adopting the Apache License v2) and Open Data support has been included in FIWARE (through CKAN GE). SmartSDK will follow the same path and contribute to the FIWARE community the newly and enhanced developed Enablers. Beyond that, it will contribute added value on top of them through: open and reusable reference architecture implementation for Smart applications based on FIWARE, open and reusable reference data models for Smart applications encoded using NGSIv2.
- ➔ **Cloudification and Microservices.** These days, new software services are embracing the cloud paradigm. In this paradigm, services can be self-provisioned by users or developers and their configuration is automatically managed to guarantee their scalability and fault tolerance. FIWARE made the development of Enablers to support cloud hosting one of its core activities. Indeed, FIWARE Lab itself is based on the FIWARE Cloud Hosting Chapter and, from FIWARE Release 4, Generic Enablers are available as Dockerized services. Even though the

¹ https://en.wikipedia.org/wiki/Representational_state_transfer

term “microservices” has been around from some years, with the advent of cloud-native applications it gained momentum. The idea of decomposing the traditional monolithic business systems into small, independently deployable services fits well with the objectives of a cloud architecture of promoting scalability, automated deployment, decentralized control, fault tolerance and resilience. The SmartSDK reference architecture will be inspired by the microservices architectural style and will take advantage of the nature of the FIWARE Generic Enablers that can be considered as atomic units of services that still deliver a business value. SmartSDK will complete this picture by providing deployable bundles of Generic Enablers that implement cloud native patterns for the development of Smart applications.

- ➔ **Market and community relevance.** Following the launch of third phase of the FI-PPP programme, FIWARE roadmap became more and more driven by market needs and by the FIWARE adopters’ feedbacks. Feedbacks from the FIWARE Accelerator programme have been a key instrument to improve the quality and market readiness of FIWARE and influenced the development of features and functionalities of FIWARE platform. Today FIWARE, with the launch of the FIWARE Foundation, becomes as a matter of fact an Open Source community where different companies and single developers can contribute following the principles regulating the community. SmartSDK will plug into the FIWARE community and go through their validation process to submit to the mainstream FIWARE community its contributions.

1.2 Structure of the deliverable

This deliverable updates the previous release of the D3.1 document. Thus, we preserved the original structure and maintained all the content that remained valid from the previous version.

- ➔ Section 2 overviews the SmartSDK Architecture Patterns, i.e. a set of basic patterns that enables the deployment of production grade Smart applications, by combining FIWARE Reference Architecture and Cloud Architecture Patterns. The section discusses how cloud architecture patterns can be applied to the different core components of FIWARE Data Management and IoT service enablement chapters. The latest revision brings content review and updates from the previous version of this deliverable, adding new features of the used tools plus the additional documentation of several new recipes developed in the covered period of work. Newly introduced content includes:
 - A set of composable PlantUML templates to generate FIWARE architecture diagrams.
 - Recipes for several FIWARE Generic Enablers (Cygnus, Comet, IoT Agents, etc)
- ➔ Section 3 presents plans for the adoption and development of FIWARE data models in SmartSDK. The section starts with a presentation of existing FIWARE data models and discuss which ones will be adopted (and hence validated) in SmartSDK application scenarios. The section overviews guidelines for the development new data models and discusses novel required models by the SmartSDK applications scenarios in relation to existing standards. Newly introduced content includes:
 - Data model validator tool.
 - Novel data models related to Alert, PollenLevelObserved, Smart POI, Smart Spot, User Context.
- ➔ Section 4 summarizes the achievements.

1.3 Audience

This deliverable is mainly intended for:

- ➔ Developers and Operators interested in deploying FIWARE Smart applications in a production

context.

- ➔ Developers and Knowledge modellers interested in adopting FIWARE data models or contributing to the initiative.

2 SMARTSDK ARCHITECTURE PATTERNS

When a software platform reaches maturity, developers look for a reference architecture to adopt it. A reference architecture provides blueprints on how to plumb the different services composing a given platform. Still, the same reference architecture can be implemented in different ways depending on the purpose of the deployed platform. For example, for a test environment you may not need the platform to scale up to thousands of requests per second, neither to be resilient to hardware failures. Concretely, this means that you can easily, for your test environment, deploy your whole architecture in a single (virtual) server.

Later, when you move the platform into production, you face additional challenges: services should be resilient to hardware failure, services should be able to scale with the number of requests, and so on. A single server is not more suitable to host your platform: you will not be able to serve requests beyond its maximal capacity and you can be sure that sooner or later it will have a failure. This means that, to provide a production level platform, you need to distribute and scale the services that compose architecture of your platform.

Distributing an architecture introduces some complexity related to the consistent management of its services. The introduction of cloud computing resulted in the simplification of deployment of distributed architectures and in the definition of the so called “cloud architecture patterns” that help developers improve the architecture of their platform to benefit from cloud computing capabilities.

FIWARE is now becoming a mature platform and developers adopting it are starting to face the problem of deploying its services into production over cloud infrastructures. While Generic Enablers reference implementations (GERis) can be easily provisioned over the Cloud (thanks to their availability as virtual machine images and Docker images), composing them to provide a resilient and scalable platform is not documented in detail.

One of the SmartSDK goals is to simplify the work of developers by providing them easy to reuse architecture patterns, that not only allow them to deploy complex architectures made of different IoT Management and Data/Context Management enablers, but also to do so in such a way that the deployment is dealing with the different production grade requirements.

This section starts providing an overview of basic FIWARE IoT Management and Data/Context Management reference architecture (i.e. without considering production requirements). Then it introduces a set of Cloud architecture patterns and discuss how such patterns can be used to extend FIWARE reference architecture to support production requirements over cloud infrastructures. Finally, it discusses how SmartSDK implements such patterns in reference recipes.

2.1 Overview FIWARE Data/Context Management and IoT Services Enablement architecture patterns

The Data/Context Management and IoT Services Enablement chapters contain the most relevant components to build so called Smart services. Thus, the work of SmartSDK focuses on them. The components of the two chapters, contrary to other ones, are also well interconnected due to the adoption of a common interface among them: FIWARE NGSIv2 [1].

The components of the two chapters are meant to be combined into a southbound (i.e. data/context producers, including IoT Devices) - northbound architecture (i.e. data processors of NGSI harmonised data). The central element in this picture is the Orion Context Broker, which allows data processors to access in different ways to data produced by sensors and other data producers.

Figure 3 presents the different components of the Data/Context Management chapter and of the IoT Service Enablement chapter as a southbound- northbound architecture. Of course, developers, depending on their needs, can select to adopt a given subset of components in their application.

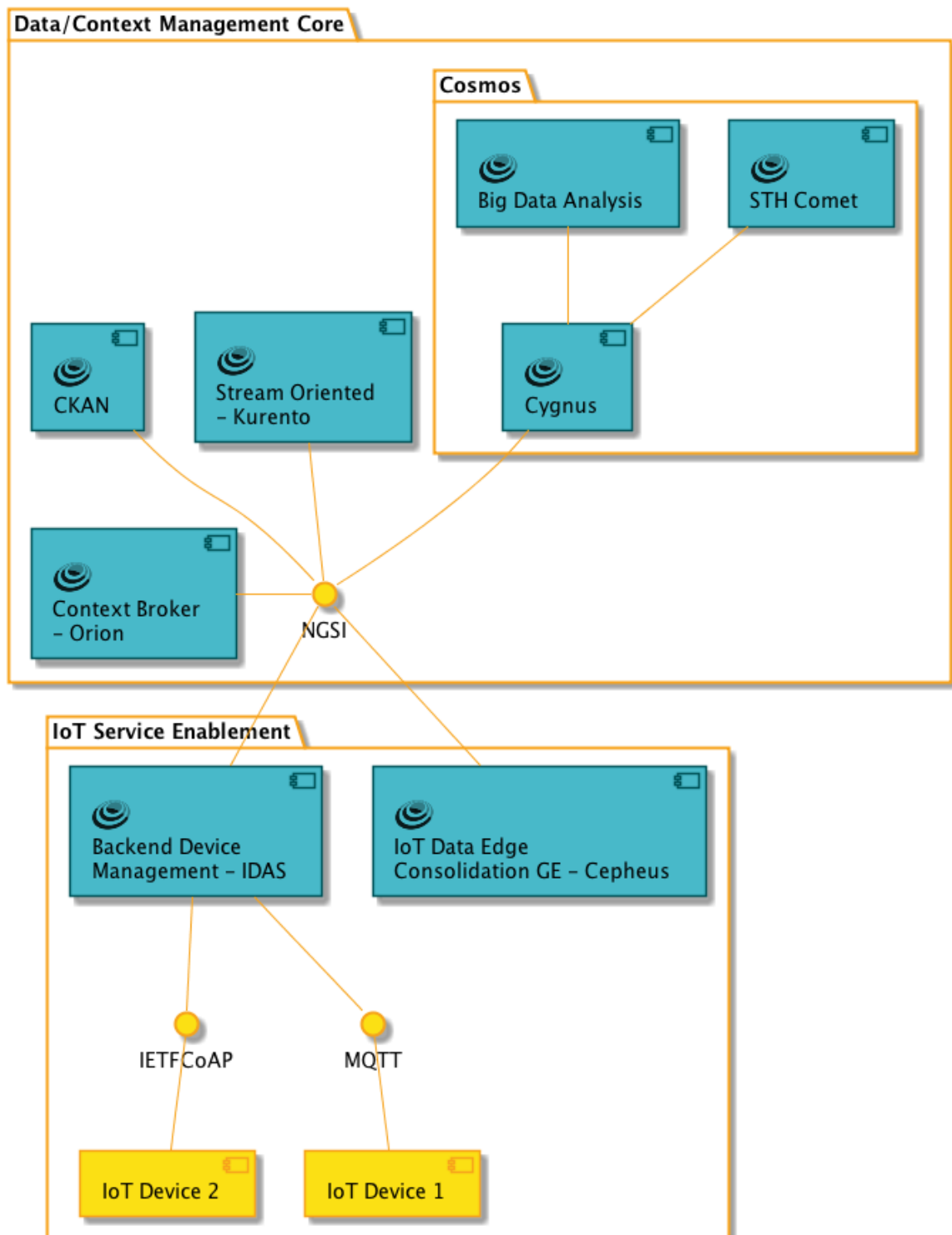


Figure 3: Data/Context Management and IoT Service Enablement Southbound-Northbound Architecture.

Core components of the Data/Context Management² chapter include:

- ➔ Context Broker - Orion, the central element of the Data Context Management architecture, provides a publish/subscribe service for context data.

² <https://catalogue.fiware.org/chapter/datacontext-management>

- ➔ Big Data Analysis - Cosmos, the solution for the analysis of NGSI data sets, made of different tools including the short time historical data storage (STH Comet) and the integration with different data stores (relational databases, big data filesystems, etc.) thanks to the adaptation capabilities provided by Cygnus.
- ➔ Stream Oriented - Kurento, an NGSI integrated multimedia processing server.
- ➔ CKAN - a repository for Open Data sets.

The IoT Services Enablement³ chapter include the following core components:

- ➔ Backend Device Management - IDAS, a set of IoT Agents which allow to register IoT devices and to transform collected data into NGSI compliant format.
- ➔ IoT Data Edge Consolidation GE - Cepheus, a solution for edge processing and aggregation of sensor data NGSI compliant.

It is worth mentioning that some of these services are meant to run only as Global Instance of FIWARE Lab. This is the case for example of Big Data Analysis Cosmos. As such, it is outside of the scope of SmartSDK to design High Available (HA) patterns for Cosmos (which, as a matter of fact, is HA and scalable in his essence for being based on Apache Hadoop⁴). Rather relevant, instead is working on simplifying the connectivity of other GEs to FIWARE Global Instances as in the case of Cosmos.

2.1.1 Support for generation of FIWARE Architecture Diagrams

To facilitate the creation of architecture diagrams based on FIWARE Enablers and SmartSDK provided extensions, we developed a set of PlantUML templates. The project code is available in github at the following url: <https://github.com/smartsdk/architecture-diagrams>

We basically defined two types of diagrams:

- ➔ UML component diagrams to define FIWARE Reference Architecture Patterns, i.e. the logical architecture that depicts the interaction among components through their interfaces (APIs). Such diagrams are encoded using PlantUML Syntax⁵.
- ➔ Directed Graphs to describe the Reference Deployment Architecture Pattern as result of the combination of FIWARE Architecture Patterns and Cloud Architecture Patterns. Such diagrams shift the focus from a logical architecture to a physical one (i.e. also covering deployment aspects of the components) and are encoded using DOT Language⁶.

Architecture Patterns are meant to be modular. Thus, you can build large patterns by composing smaller ones. Such graphs can be easily rendered online with gravizo⁷ or on a Linux computer with graphviz⁸.

The online repository includes several examples, here we present a basic example as a guideline for the development of additional diagrams. The example assumes a pattern that includes a FIWARE GE (e.g. ContextBroker) and a SmartSDK GE (e.g. NGSI Timeseries) that are interfaced through a NGSI API.

To specify that a component is developed by FIWARE, you can use the following stereotype:

³ <https://catalogue.fiware.org/chapter/internet-things-services-enablement>

⁴ Readers interested to learn about deploying Hadoop in High-Availability can refer to [2].

⁵ <http://plantuml.com/component-diagram>

⁶ <http://www.graphviz.org/content/dot-language>

⁷ <http://g.gravizo.com>

⁸ <http://www.graphviz.org>


```
FIWARE(timeseries,"NGSI TimeSeries",component)
```

To specify that a component is developed within SmartSDK, you can use the following stereotype:

```
SMARTSDK(timeseries,"NGSI TimeSeries",component)
```

Accordingly, you can inject such components in a PlantUML diagram as follow:

```
@startuml;

skinparam componentStyle uml2

!define ICONURL https://raw.githubusercontent.com/smartsdk/architecture-
diagrams/smartsdk-template/dist
!includeurl ICONURL/common.puml
!includeurl ICONURL/fiware.puml
!includeurl ICONURL/smartsdk.puml

package "Example of Reference Architecture Pattern" as example {
    interface NGSI
        FIWARE(ctx,"Context Broker \n - Orion",component)
        SMARTSDK(timeseries,"NGSI TimeSeries",component)
        NGSI -left- ctx
        NGSI -right- timeseries
    }
@enduml
```

2.2 Overview of Cloud Architecture Patterns

Throughout the history of software development projects, specific software engineering practices have been proved successful to solve certain recurring problems. When developers started noticing they were applying the same kind of solutions for certain types of problems, these solutions were abstracted from the specific problem and defined as “Patterns”.

Patterns are regarded as well-defined, well-known and maybe most importantly, reusable solutions for common software development problems. They can emerge from the lower level of software implementation details, as in the case of Design Patterns, or from the software deployment structure and the relationship of its main components. We will refer mostly, though not exclusively, to the latest form, commonly known as Architecture Patterns.

The following is a simple summary of the most relevant patterns in the context of cloud-native applications. Its purpose is to give a better understanding of the patterns by exemplifying some of them. Interested readers of this subject can refer to literature such as Design Patterns [3] or Cloud Architecture Patterns [4].

2.2.1 Basic concepts related to Distributed Systems

As mentioned in the introduction of Section SmartSDK Architecture patterns, moving from a centralised to distributed architecture poses some challenges. Most of the challenges relates to the fact

that often a service is not *stateless* and distributing the service on multiple nodes implies also that the status must be distributed across the nodes.

2.2.1.1 Stateless versus Stateful services

A service is *stateless* if at any point in time the value of its output(s) depends only on the value of the input(s). On the other hand, a service is *stateful* if at any point in time the value of the output(s) depends on the value of the input(s) and of an internal state.

In cloud computing platforms, stateless services are ideal for scaling horizontally⁹ and for high-availability, since they can be deployed multiple times without any interdependency; whereas stateful services are more suitable for scaling vertically¹⁰. High availability and horizontal scaling, in the case of stateful services, require ways to recover or synchronize the status of the system.

2.2.1.2 CAP Theorem

In the case of distributed stateful services, solutions to synchronize the status of the service across all the nodes need to be applied to guarantee that the distributed service acts coherently as a single service. In the context of a reliable and resilient distributed system, we can identify the following desirable properties:

- ➔ Consistency: All nodes composing the service have the same status at the same time.
- ➔ Availability: The service can provide a reply at any given time.
- ➔ Partition tolerance: The service continues to work even though some of the nodes composing it are not reachable temporarily or definitively.

Unfortunately, Brew's conjecture [5], better known as CAP Theorem, states that a distributed system can have simultaneously only two out of three of the above properties. While of course the theorem is valid under certain definition of Consistency, Availability, and Partition Tolerance, it is generally valid. As a matter of fact, distributed systems are built on networks and networks aren't completely reliable. This implies that distributed systems must tolerate partition. According to the CAP theorem, this means we are left with two options:

- ➔ **Availability and Partition Tolerance.** Most of the data replication solutions on the market today adopt Availability and Partition Tolerance as properties and implement mechanism to ensure that the system reconciles in a short time, minimizing the time window where an observer would find the status of the system inconsistent. These systems are generally called *eventually consistent*.

⁹ Horizontal scalability is the ability to increase the capacity of a service by running more instances of the same service.

¹⁰ Vertical scalability is the ability to increase the capacity of a service by increasing the number of hardware resources (e.g. CPU, RAM) allocated to the service.

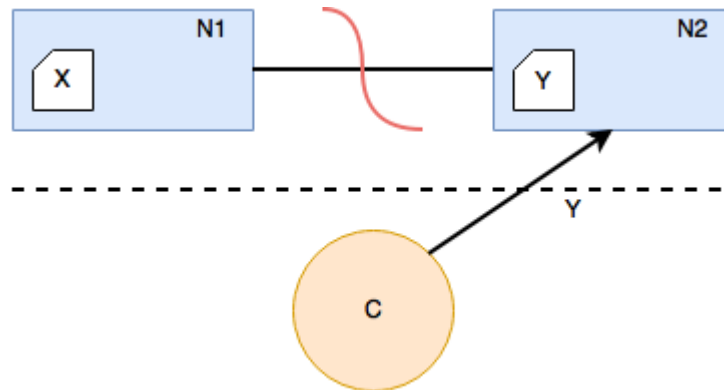


Figure 4: Availability and Partition Tolerance.

- ➔ **Consistency and Partition Tolerance.** Solutions based on Consistency and Partition Tolerance, basically implements atomic reads and write. The system will return a response only when the global status is consistent (and hence if the system is not partitioned). If for any reason a node is not reachable, the system will return no response (or an error). I.e, the system is *eventually available*.

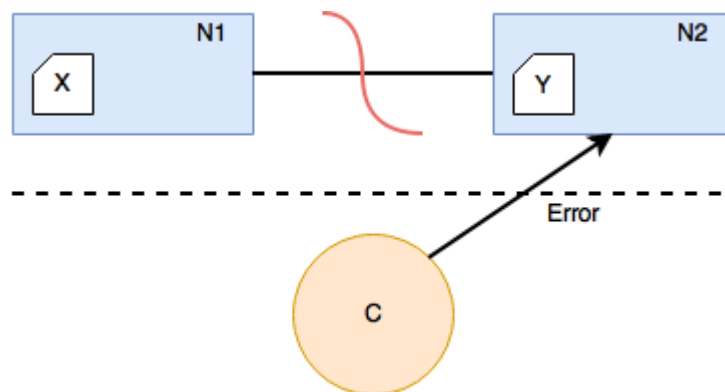


Figure 5: Consistency and Partition Tolerance.

2.2.2 Scalability Pattern

The main principle of the Scalability Pattern is that applications should be prepared to easily adapt to a sudden change in its demand of the resources. Scalability is therefore a measure how well an application can adapt to, for example, serve more users.

Applications may scale horizontally or vertically. Ability to scale horizontally is essential to support the distribution of an application across multiple nodes and to ensure that the scalability of a service is not constrained by the capacity of the server on top of which the application may run.

2.2.3 High Availability Pattern

Imagine a Bank was running their Home Banking service on a single server and suddenly, something bad happened to that server. For example, the computer overheats and turns down. If the application was not prepared to react to this scenario, Bank users would instantly lose access to the service.

The main idea behind this pattern is that applications should be resilient to hardware failures and have either an extra instance of the application ready to cover the demands of the failed instance or enough resources in the running instances to distribute among them the load the failed instance was attending. Figure 6 shows an example of a deployment of an App which must attend 10M users. Since each server can serve up to 5M users, the deployment on 3 servers can be considered in HA with tolerance

to 1 server failure.

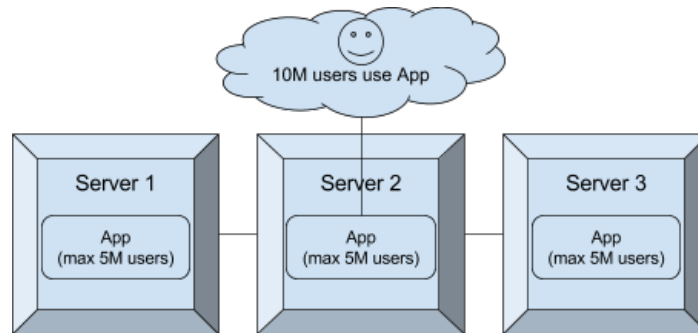


Figure 6: HA Deployment.

2.2.4 Multi-site pattern

Imagine that Youtube had its services and content running only in California. If this was the case, users in the other side of the world would experience a terribly slow service because of the network latency perceived due to the distance data would need to travel in the network.

In order to give all users a similar latency and hence quality of service, the pattern (shown in Figure 7) suggests that software should actually be deployed in servers located in different points of the globe. Of course, a synchronization mechanism among sites should be adopted so that the content is shared, cached and seen relatively with an equal quality everywhere.

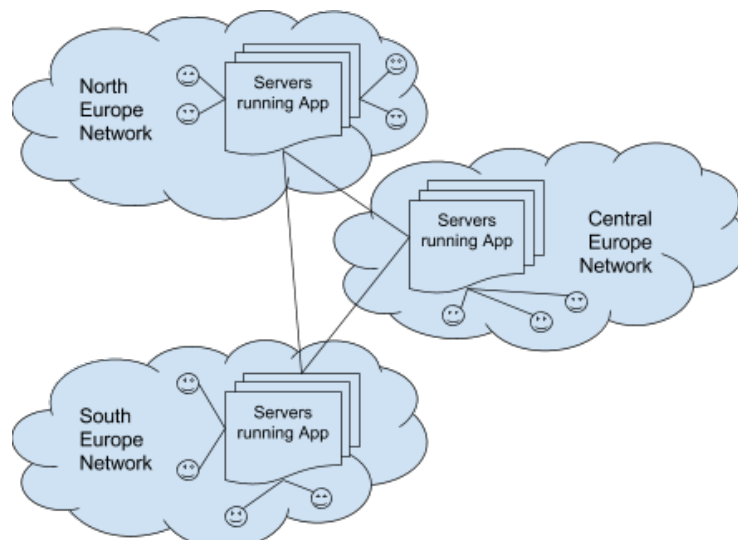


Figure 7: Multisite deployment.

2.2.5 Co-locate Pattern

This pattern complements the Multi-Site and suggests that services that usually work together depending on each other should be deployed to the same server so as to reduce the delays produced by communication. Packet transmission times in network connecting different servers or even worse different data centres can severely affect performance.

2.2.6 Queue-Centric Workflow Pattern

This pattern, illustrated in Figure 8, decouples the communication between services that “consume” data and services that “produce” data and is particularly useful when the consumption and production layers (as often happens) have different performance. The usage of a queue between the two layers acting as a buffer enables asynchronous communication and let each part run at its own pace (within a

constrained production/consumption ratio, of course).

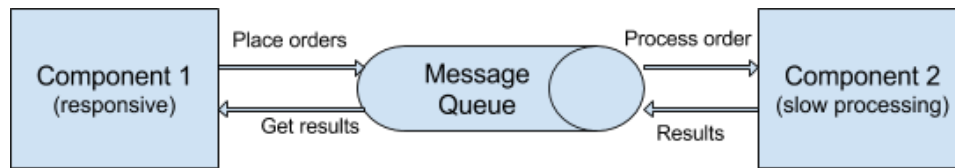


Figure 8: Queue-Centric Workflow.

2.3 Methodology

So far, we have listed some FIWARE Architecture Patterns, i.e. how FIWARE services can be combined to define a FIWARE reference platform for IoT and Data Management. Also, we have reviewed common Cloud Architecture Patterns, particularly how services can benefit from cloud architecture principles to provide higher scalability and resiliency.

The next step is to combine the two ideas to apply Cloud Architecture Patterns to FIWARE Architecture Patterns and provide solutions for deploying FIWARE IoT and Data Management in a scalable and resilient way.

Each FIWARE GErI has its own architecture, so this means that there is no single solution that fits all needs. Thus, each FIWARE GErI needs to be analysed in order to understand how to apply Cloud Architecture Patterns to its architecture.

The methodology we follow in the definition of the “SmartSDK Architecture Patterns” (i.e. the patterns that combine FIWARE and Cloud Architecture Patterns) is based on a simple principle: composability. This principle states that the definition of patterns is guided by the idea that we break down a larger component in smaller ones, so that recurring patterns can be easily reused. For example, MongoDB is used in different FIWARE GERIs as a backend solution, so we can define a specific pattern for MongoDB that then can be reused by other patterns.

The methodology works as follows:

1. Analyse the GErI architecture and identify its main components;
2. For each component identified, understand if it is stateful or stateless;
3. For each identified component, apply Cloud Architecture Patterns according to its type (stateful vs stateless) and supported solutions for distributing its state across multiple nodes;
4. Document the architecture pattern of the different components composing the GErI;
5. Compose the overall architecture pattern for the GErI from the sub-patterns;
6. Translate the defined patterns into deployable recipes.

Of course, it may not be possible to apply Cloud Architecture Patterns to all GErI. Some GErI may have architectures that are not suitable for cloud environments because, for example, do not provide ways to expose part of the status and share it across multiple nodes. It is outside the scope of SmartSDK to modify the code of such GERIs to enable the application of Cloud Architecture Patterns to them. Nevertheless, the analysis has already generated feedback to GErI owners regarding the limitations of their current architecture, for example in the form of github issues and or as debates in the SmartSDK presentations held at the different FIWARE summits.

2.4 SmartSDK Architecture Patterns

Many of the ideas explained so far regarding Architecture Patterns can be applied to different FIWARE Generic Enablers. In this section, we will highlight how some Generic Enablers would benefit from the application of the patterns and which considerations must be considered for the

applicability of those. In particular, we have chosen the most used generic enablers in the SmartSDK applications scenarios, which are also among the most popular GEs in the FIWARE community.

2.4.1 Context Broker

The Context Broker lies at the core of the Data Management chapter of the FIWARE Architecture and its reference implementation is called Orion. Orion is basically composed of two complementing parts, namely a front-end and a backend.

The frontend is the Orion service implementing the NGSI REST API providing users features such as queries, subscriptions and notifications for context data. This frontend is stateless, in the sense that there is no state being persisted on the service instance and hence if an instance breaks, no data is lost.

On the other hand, the backend is the stateful part of Orion which gives persistence to its data, up to a certain point. It is usually implemented as a MongoDB database running in a different process and known to Orion as the “db”.

2.4.1.1 Scalability

A simple Orion configuration with one service in the frontend and one in the backend will be able to attend a certain maximum number of users. This limit, known as “capacity” is usually dictated by both hardware and software characteristics of the deployment. Moreover, such scenario is not well-suited for failure resiliency as explained in previous sections, in the sense that if one of the parts fails, the whole service falls apart.

To either increase the capacity of the service or scale it down (unused resources in the cloud usually mean wasted money), the administrators can scale both frontend and backend separately. However, there are important considerations to keep in mind.

The frontend, as a stateless service, can be scaled up or down without worrying about data losses. The only thing to keep in mind is that those services going down should not have been directly exposed to the public using individual details (e.g specific IPs). This is to avoid users attached to the fallen service not being able to reach the service anymore. To avoid this, a common practice, as shown in Figure 9, is to put a load balancer in-front of all the frontend services to act as an entry-point whose connection details remain the same. Another DNS-based solution would be to dynamically associate the IPs of the frontend to a specific Orion service name, which clients would interact with.

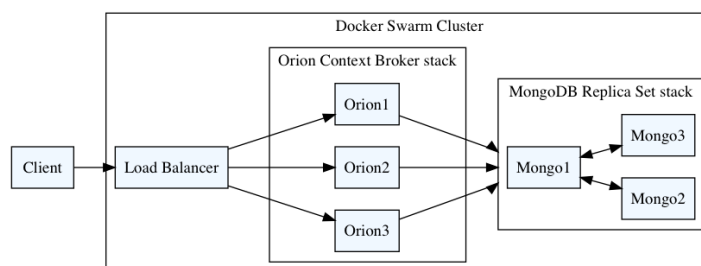


Figure 9: High Available and Scalable Orion Context Broker Architecture Pattern.

Nevertheless, scaling the backend requires a more sophisticated approach, because data should be the same across all instances to guarantee users always see the same set of data regardless of the service responding their requests. For example, when data is added to one of the instances, this new information should be replicated to the rest of the instances as soon as possible.

In general, changes in the data introduce inconsistency into the system that needs to be sorted out. To solve this, distributed algorithms based on the quorum principle exist so that instances can reach consensus and keep the same data across the dataset. These algorithms, however have certain requirements that should be met. For example, in MongoDB version 3.0 there is a maximum of 50 members out of which a maximum of 7 can be voters. Also, there must be an odd number of voting

members in order to have a tie-breaker in the case of tied elections. Further details can be read in the Replication Manual of MongoDB¹¹.

2.4.1.2 High Availability

An administrator of the Orion service, knowing the capacity of the current service and having the ability to scale up both frontend and backend, will be able to run the Orion service in High-Availability. There are, though, two more things required to do so.

Scaling up the frontend services would be useless without the ability to actually distribute the incoming requests among them. This is the main task of the Load Balancer, an essential element in the HA architecture.

Also, in the case of failures, if the services keep on failing one after another at some point there will be no more redundancy of services and Orion will be down. Consequently, a reconciliation mechanism instrumented to recover the failed instances will be necessary. This could be either manual (i.e, an administrator re-launching a failed instance) or a software-based automated solution.

2.4.1.3 Multisite

This pattern becomes relevant only for huge deployments of Orion across many geographical areas. The benefits for simpler or smaller deployments of Orion will not outweigh the challenges of this kind of deployment.

The multisite deployment of the frontend might not be technically as challenging as the backend, but still both would require significant efforts. The frontend would require configuring load balancers to send requests to different data-centres based on physical proximity of the users to the Orion instances.

The backend, on the other hand, would require a robust replication mechanism in order to keep data consistency while performing within reasonable time limits, considering the distances data will have to travel. In such a distributed scenario, database administrators should consider the per-region data availability needs because not all data may be needed in all regions and hence a sharding scheme may be more suitable. This means, some data are better kept in only certain regions, not necessarily in all.

2.4.2 Comet Short Term History (STH)

Comet is a component of the FIWARE ecosystem in charge of storing and retrieving historical raw and aggregated time series information about some context data; for example, entity attribute values registered in an Orion Context Broker instance.

Comet interacts with an NGSI data generator such as Orion Context Broker via notification mechanisms and requires a database, typically a MongoDB, to store raw and aggregated data from these notifications.

In a scenario consisting of heavy loads of notifications, having a single instance of Comet to attend and process all the data could easily become a bottleneck for the overall architecture. It is therefore desired to have a receptive frontend with a distributed load of notifications as show in Figure 10 below.

¹¹ <https://docs.mongodb.com/manual/replication/>

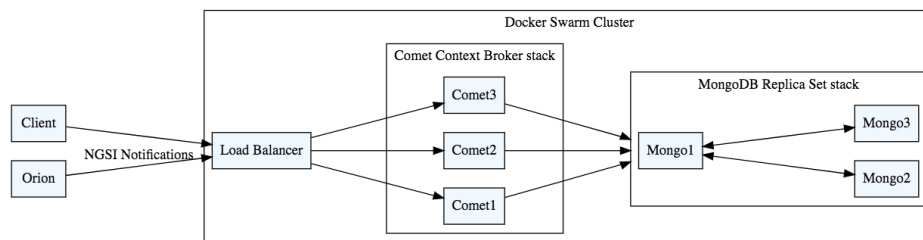


Figure 10: High Available and Scalable Comet STH Architecture Pattern.

2.4.2.1 Scalability

Due to the architectural similarities with the aforementioned Context Broker, the same considerations for scalability apply to Comet both for the front-end as well as the back-end.

2.4.2.2 High Availability

Due to the architectural similarities with the aforementioned Context Broker, the same considerations for high availability apply to Comet both for the front-end as well as the back-end.

2.4.2.3 Multisite

Deploying Comet STH in a multi-site docker cluster will require the same considerations mentioned in section **Error! Reference source not found.**

2.4.3 Cygnus

Cygnus is a connector in charge of persisting certain sources of data in certain configured third-party storages, creating a historical view of such data. In particular, we will be working with the “cygnus-ngsi” version of Cygnus. The simplest deployment of Cygnus is depicted in the image below.

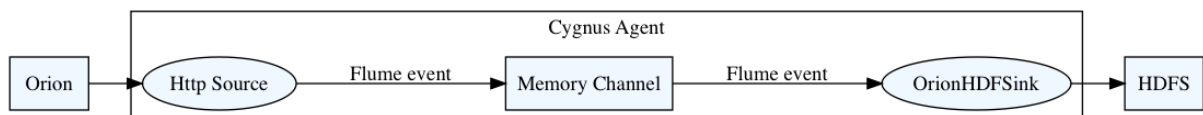


Figure 11: Cygnus Architecture Pattern.

As it can be seen from the Figure 11, the main component of the architecture is the Agent, whose source consumes notifications from an NGSI origin such as Orion and dispatches those events to channels, each of which is connected to some sort of sink. Cygnus recipes should simplify for users the process of their agent’s deployment with different types of sinks.

2.4.3.1 Scalability

Scaling Cygnus can be done both at the level of the agent and/or at the level of the sink database.

At the level of the agent, both the source and sink are stateless in the sense that are services which just transform and put or take data out of a channel. The channel, in any of the two versions (memory- or file-based) is stateful for a short period of time (until data is moved by the sink to the definite storage solution). With that consideration in mind, for the scalability discussion, this channel can also be considered stateless and be scaled with docker by changing the number of replicas.

At the level of the database sink, again, each database will have its own scalability mechanism and no solution can be generalised for all persistence solutions.

2.4.3.2 High Availability

As explained in its official documentation¹², Cygnus provides users the possibility to configure its internal architecture applying patterns such as the Queue-centric workflow to improve its performance for some specific scenarios. However, in order to improve the reliability of the Cygnus deployment applying the High Availability pattern, it is required to deploy multiple agents at the same time. The recipe available in the SmartSDK repository allows users to easily deploy multiple agents, as show in Figure 12 below.

As mentioned in the previous section, the highly-available deployment of backend database varies significantly with regards to the solution (technology) used for the database and hence it is up to the administrator deploying the service to achieve this

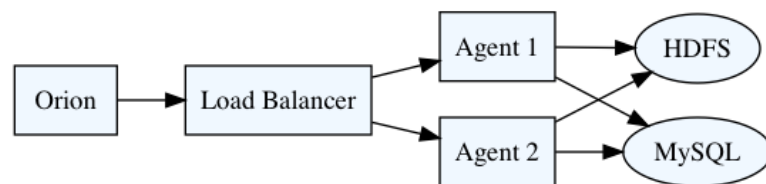


Figure 12: High Available and Scalable Cygnus Architecture Pattern.

2.4.3.3 Multisite

Deploying Cygnus in a multi-site docker cluster will require the same considerations mentioned in section **Error! Reference source not found.**

2.4.4 IDAS

IDAS is the reference implementation of the Backend Device Management Generic Enabler¹³. Its main usage is the Typical IoT use-case Scenario I: Common Simple Scenario of the overall FIWARE IoT architecture¹⁴.

As of now, IDAS can be seen as a group of IoT Agents supporting different protocols on top of different transport options. Some of the supported protocols for the different available Agents are UL2.0, JSON, OMA-LWM2M and SIGFOX. It could be said though, that an average Agent deployment architecture typically looks like Figure 13.

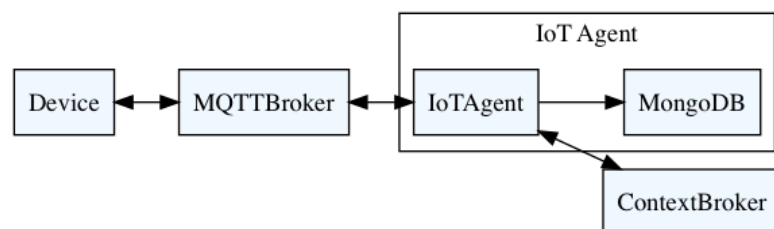


Figure 13: IDAS Architecture Pattern.

¹² <https://fiware-cygnus.readthedocs.io/en/latest/architecture/index.html>

¹³ https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FIWARE_ArchitectureDescription.IoT.Backend.DeviceManagement

¹⁴ https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/Internet_of_Things_IoT_Services_Enablement_Architecture#Typical_IoT_use-case_Scenarios_.28I.29:_Common_Simple_scenario

2.4.4.1 Scalability

Scaling IoT Agents may become a requirement when multiple devices with their per-zone MQTT Broker are deployed in multiple zones, hence creating a significant load on the IoT Agent instance typically deployed in the cloud, close to Orion Context Broker.

Scaling the IoT Agent can be done independently from the backend database, provided that all Agent instances run with the same configuration. As it will be explained in section 2.5.3.6, this can be easily achieved with Docker.

Scaling the database, which represents the stateful part of the Agent, requires the same considerations mentioned for Orion Context Broker's backend, because both use the same MongoDB backend of typically one instance that can be scaled to multiple through the construction of a replicaset.

2.4.4.2 High Availability

To improve its resilience, multiple instances of the internal services (IoTAgent and MongoDB) could be deployed in the way presented in Figure 14.

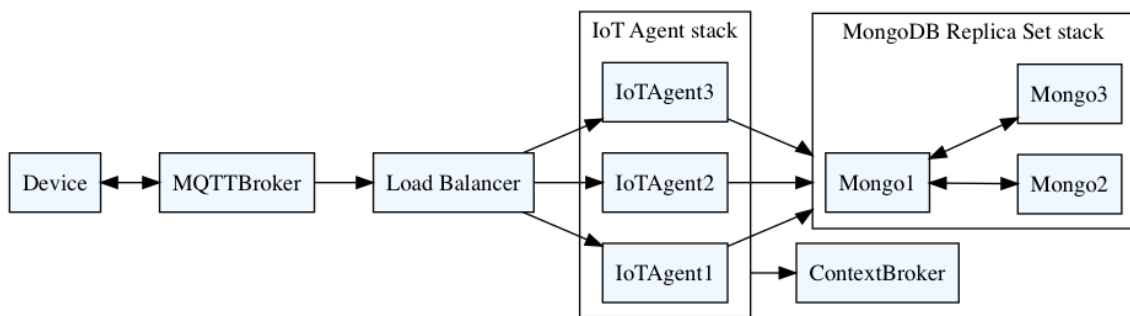


Figure 14: High Available and Scalable IDAS Architecture Pattern.

2.4.4.3 Multisite

Deploying IDAS in a multi-site docker cluster will require the same considerations mentioned in section **Error! Reference source not found.**

2.4.5 CKAN

A basic deployment of the CKAN Generic Enabler consists of the deployment of group of different services, some of which are stateless whereas others are stateful. The diagram in Figure 15 shows the architecture of a simple deployment of these services.

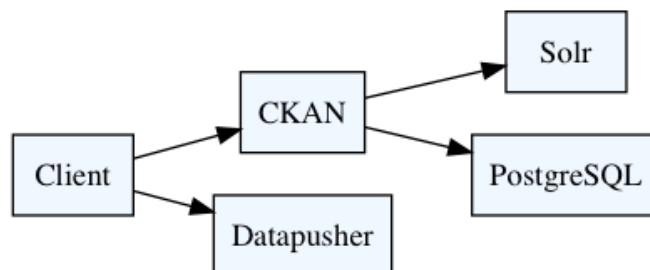


Figure 15: CKAN Architecture Pattern.

2.4.5.1 Scalability

The scalability of the CKAN GE could be approached first with a multiple-instance deployment of the stateless services, namely CKAN and Datapusher. A load balancer in front of each stateless service would distribute requests among the multiple replicas of the service. This would make the architecture look something like in Figure 16 below.

2.4.5.2 High Availability

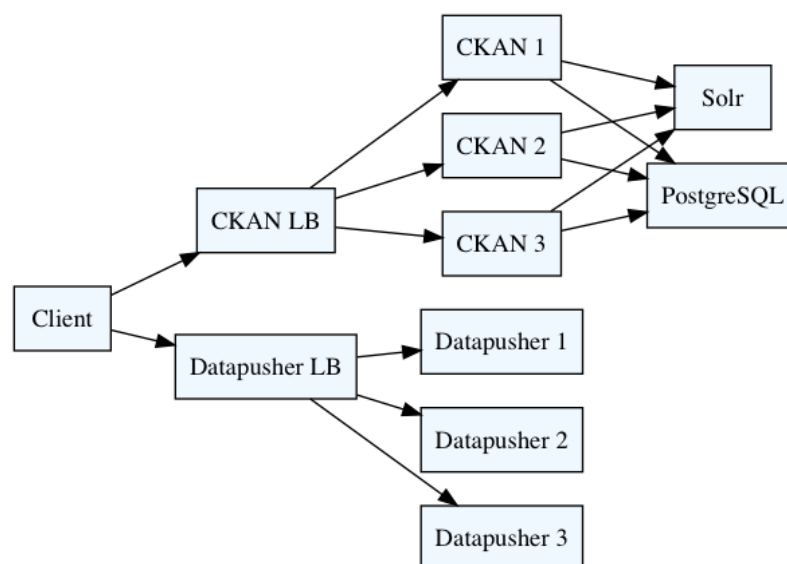


Figure 16: High Available and Scalable CKAN Architecture Pattern.

The replication of the stateful services “Solr” and “PostgreSQL” would require a more careful configuration. Some hints about how to run a distributed solr are mentioned in the official Solr’s Docker image page¹⁵ whereas the replication of the “PostgreSQL” database is covered in the replication wiki page¹⁶.

2.4.5.3 Multisite

Deploying CKAN in a multi-site docker cluster will require the same considerations mentioned in section **Error! Reference source not found.**

2.4.6 QuantumLeap

QuantumLeap is a new service developed in SmartSDK Project, used to persist historical values of NGSI Context Data. In its essence, it consists of two essential type of services, namely the frontend python-based service named “QuantumLeap” and the backend database, where data is actually persisted. The main implementation of the backend uses Crate as the default database, as it can be seen in Figure 17.

Complementary services such as grafana, redis and OSM will be left aside for this discussion, as they do not represent core services of QuantumLeap, though a similar analysis could be done of course.

¹⁵ https://hub.docker.com/_/solr/

¹⁶ https://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling

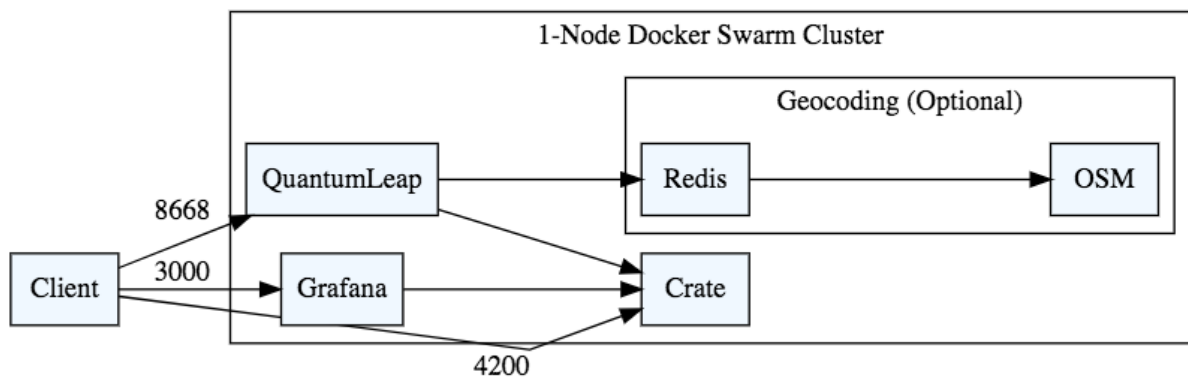


Figure 17: Simple deployment of QuantumLeap.

2.4.6.1 Scalability

Scaling QuantumLeap frontend with docker is straightforward as it is a stateless service, so any number of replicas would work fine simultaneously.

On the other hand, scaling the Crate database requires some adjustments. To begin with, and similarly with the situation in different databases, it makes more sense to keep no more than one instance per available node in the cluster. This is because, in order to provide persistence to data, a local volume needs to be mapped to the node, and unless a complex network volume solution is being used, having multiple replicas mapping to the same volume in a node is a recipe for chaos.

Another requirement for the scalability of the Crate cluster, is that, as a shared-nothing cluster architecture Crate has, the service responsible for forming the cluster needs to reach out to all the available instances in the cluster (their IPs). This is a new requirement that, in the context of docker swarm, will be attended by customising the deployment endpoint mode and the introduction of a second load balancer, to persist external admin accessibility to the cluster configuration. This new scenario is depicted in **Error! Reference source not found.**

2.4.6.2 High Availability

A high-available deployment of QuantumLeap core services is shown in **Error! Reference source not found.** As mentioned earlier, maintaining multiple instance of the QuantumLeap service becomes quite an easy task with docker because it is a stateless service. On the backend side, Crate supports the automatic formation of clusters with a shared-nothing architecture to support HA deployments. The only caveat to consider in a default docker swarm environment is that, in order to allow the retrieval of all the IPs of the nodes forming the cluster, the service needs to be deployed bypassing the routing mesh using DNSRR instead of a virtual IP.

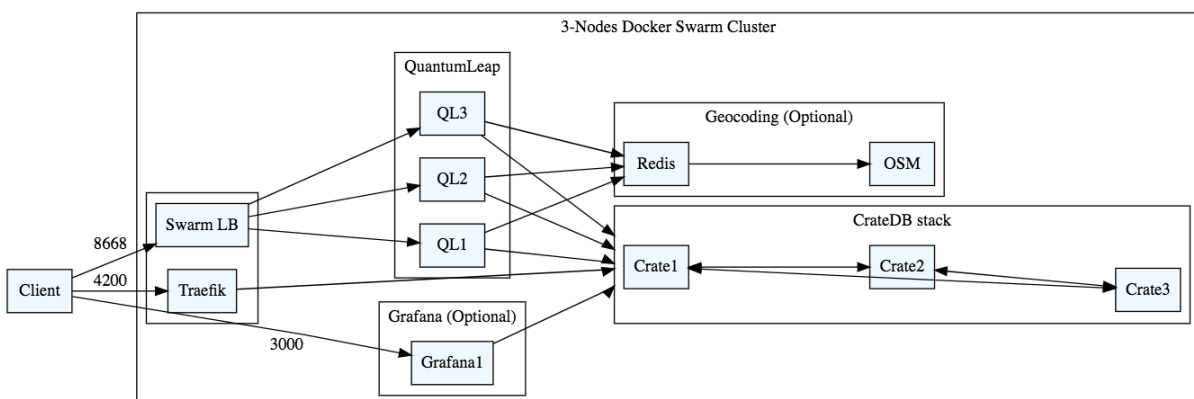


Figure 18: HA and Scalable deployment of QuantumLeap

2.4.6.3 Multisite

Deploying QuantumLeap in a multi-site docker cluster will require the same considerations mentioned in section **Error! Reference source not found.**

2.4.7 API Umbrella

API Umbrella is a relatively new addition to group of components in the FIWARE Security Chapter. It allows user to protect and monitor the usage of their APIs behind this proxy (“umbrella”). Its architecture is very well documented in corresponding website¹⁷. A simple deployment diagram of this service is shown in Figure 19.

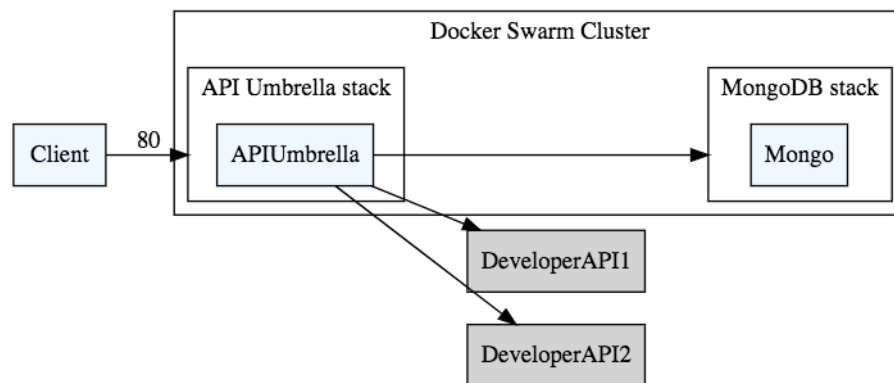


Figure 19: Simple deployment of API Umbrella w/o analytics.

2.4.7.1 Scalability

The scaling the backend of the deployment could be done as explained in section 2.5.3.1, because this service also uses MongoDB as its backend. However, at the time of writing there is a special consideration to be taken, and it is the fact that the current implementation of API Umbrella is not using drivers with proper support for mongoDB replicas discovery and hence the configuration of the service requires the user to define ad-priori the IPs of the services that will take part in the replicaset. In the context of docker swarm clusters, this is not ideal nor reliable because IPs could be changing through the lifespan of the service execution. We have documented these considerations in the recipe and included configurable scripts to inject those values in the docker-based deployment.

On the other hand, the frontend up and down scaling can be achieved as explained in section 2.4.1.1. Like the Context Broker, the API service can be considered stateless and scaled without data loss concerns.

2.4.7.2 High Availability

A high-available deployment of API Umbrella, without the analytics package, is depicted in Figure 18. It resemblances many similarities with other schemas already shown, which brings many opportunities and benefits for the deployment recipes in reusability terms.

¹⁷ <https://apiumbrella.io/>

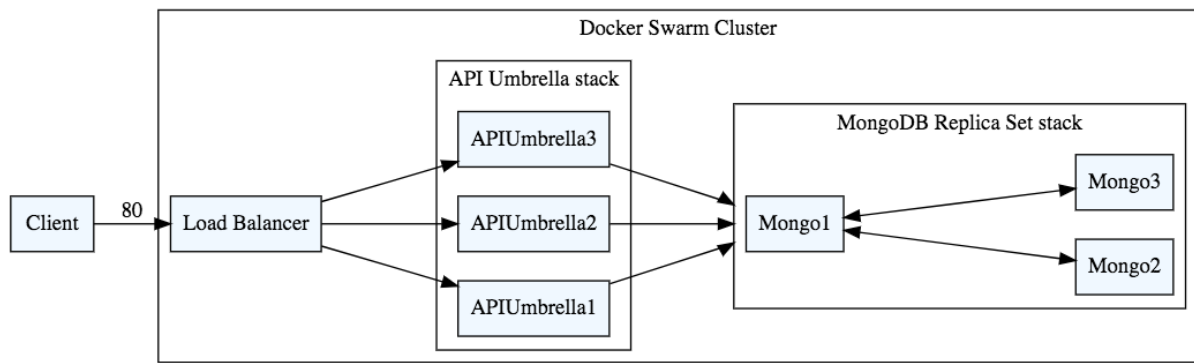


Figure 20: HA and Scalable deployment of API Umbrella w/o analytics.

2.4.7.3 Multisite

Deploying API Umbrella in a multi-site docker cluster will require the same considerations mentioned in section **Error! Reference source not found.**

2.5 Realization of Architecture Patterns recipes in SmartSDK

As shown in previous section, Architecture Patterns are powerful tools that can be applied to make software applications more robust. In line with its goal of simplifying and improving the development of FIWARE Applications, SmartSDK aims to include these patterns in its toolbox in the form of recipes.

FIWARE Applications are typically developed with the usage and composition of different FIWARE Generic Enablers (GE). A simple look at the documentation of complex GEs such as Cygnus¹⁸ or even at the composition of different GE such as the cases mentioned in the page for Data Management Architecture¹⁹ shows that there is room for applying software architecture patterns.

GEs are relatively complex pieces of software running as services, developed in either one or, most likely, multiple programming languages. As such, these services require a fair amount of configuration and pieces from other systems to work. Furthermore, many times services need to be re-deployed to different places, and hence must be able to run in different platforms. All these challenges are not specific to FIWARE, but rather common in the software industry. To overcome these difficulties, services are nowadays being containerized.

There are many technologies supporting containerization nowadays, the most prominent one in the market being Docker. On top of the containerization technology, orchestration systems are built in order to control and manage the lifespan of containers. Orchestration typically involves scheduling, launching and monitoring containers running in different nodes of a cluster.

There are many orchestration technologies in the market nowadays, each with its pros and cons, and the most popular are Kubernetes, Swarm and Mesos/Marathon. Among the three, Swarms stands out as the simplest and easiest to use. Moreover, it is developed by Docker, so it integrates quite well with the tooling of the Docker ecosystem, making it the easiest to adopt for people already working with Docker containers. It is true though that both Mesos/Marathon and Kubernetes are both older and more robust systems than Swarm in terms of offered features for containers orchestration and scaling capabilities. Nevertheless, it is also well known that this power comes with the cost of bigger complexity and harder learning curves.

Docker has been the chosen container solution in the FIWARE community. In line with this decision,

¹⁸ <http://fiware-cygnus.readthedocs.io/en/latest/architecture/index.html>

¹⁹ https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/Data/Context_Management_Architecture

and the previously mentioned advantages, SmartSDK decided to adopt all the technologies of the Docker ecosystem, including Swarm. SmartSDK acknowledges some features are still missing in Swarm, as it will be pointed out in later sections of this document. However, considering Docker's fast growth²⁰, we see the distance with its competitors is being shortened as time goes by.

The following subsection will help the reader understand how the patterns will come to life with the underlying usage of Docker-based technologies.

2.5.1 Foundation

2.5.1.1 Docker Compose

Deploying containers is easier than manually handling custom deployed services on either virtual machines or bare metal. However, when applications grow in complexity and they start being divided into multiple services, handling multiple containers can become a very challenging task.

To simplify the orchestration of containers, Docker introduces a native tool called Compose²². Among the main characteristics that make Compose attractive to the audience of SmartSDK users, we could highlight the following three.

- ➔ **Ease of use:** With Docker Compose, users can launch multiple services properly configured with the ease of a single command line.
- ➔ **Declarative approach:** Compose adopts a declarative approach for the configuration, which can be read in a simple script file. This is much easier than having to dig and mine multiple files of code to customize certain parameters of the application.
- ➔ **Can be easily reused:** This brings the power of writing once and reusing multiple time, allowing users to attend new use-cases by combining pre-scripted recipes.

Below, a simplified example of a Docker Compose file:

```
version: '3'

services:
  mongo:
    image: mongo:${MONGO_VERSION}
    command: --nojournal
  orion:
    image: fiware/orion:${ORION_VERSION}
    ports:
      - "1026:1026"
```

With such a file, a user can simple execute a command like “docker-compose up” and have all the stack of services up and running, showing how easy-to-use compose is.

Also, the declarative approach of the file simplifies both understanding and customization. For example, any reader could tell how many services are being deployed (2 in this example) and which port is being used by the Orion service (1026).

Finally, if a user wanted to use Docker compose for their microservices-based application using Orion,

²⁰ <https://www.docker.com/docker-news-and-press/docker-automates-and-democratizes-container-orchestration>

²¹ <https://dzone.com/articles/docker-swarm-lessons-from-swarm3k>

²² <https://www.docker.com/products/docker-compose>

they could reuse the same file removing the need to redefine Orion's configuration in their custom file.

Compose has been adopted by many FIWARE Generic Enablers for simple scenarios such as the Tour Guide²³ and will be adopted by SmartSDK for more complex recipes. However, SmartSDK requires a special set of features available only from Docker Compose version 3, as explained in the following section.

2.5.1.2 Docker Swarm

Swarm is Docker's native tool for defining, configuring and managing clusters of Docker hosts. It started as a standalone application but since the release of Docker 1.12 a new Swarm, known as "swarmkit" or "swarm mode", was integrated into the Docker engine. This last one, swarm mode, is the Swarm²⁴ this document will be referring to.

However, Compose was not ready to fully integrate with the new Swarm until the release of Docker version 1.13. However, docker 17.06 is the first Docker version supporting Docker Compose files version 3.3, giving users the features of Docker Compose mentioned in the previous section with the power of the new swarm features explained further below. Therefore, it was decided to support Docker versions from 17.06 in SmartSDK. Due to this commitment to follow the state of the art of the Docker releases, as the project evolves, we may push the minimum required version to a higher value, of course, always respecting the official available releases so that users do not need to "customize" installations to beta versions.

The reader will note that Docker introduced a new scheme for naming the software versions and their products. However, all comments are still valid and the developed recipes work with the newer versions. For an overview of the releases, please refer to the release notes page²⁵.

Before we mention the new Swarm features SmartSDK is taking advantage of, it is worth explaining what a Swarm is in the first place. As it can be seen from Figure 21, a Docker Swarm is basically a collection of nodes (usually called a cluster). Nodes are machines running Docker which could be physically distributed in datacentres of different providers (such as AWS or FIWARE lab) but must be interconnected through the same network. On top of the cluster, users can therefore deploy their stacks (i.e., collection of services) and, according to the deployed stack and related constraints, Swarm will orchestrate the services part of the stack and the nodes part of the cluster, providing the needed synchronization and communication between them.

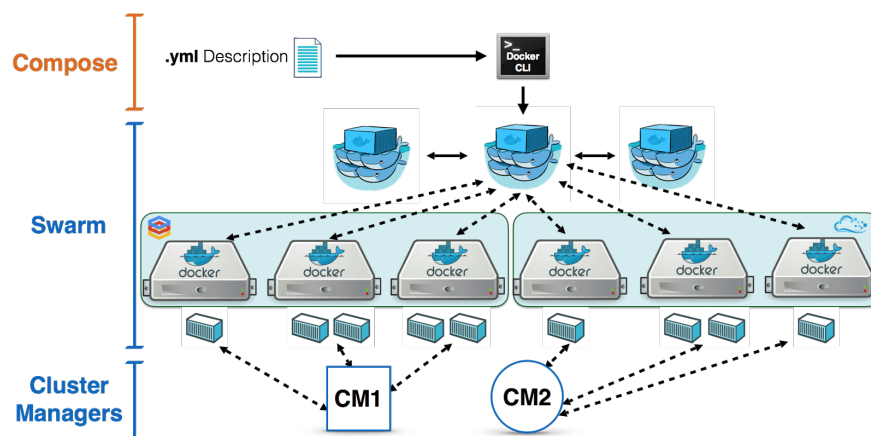


Figure 21: Docker Swarm²⁶.

²³ <http://fiwaretourguide.readthedocs.io/en/latest/>

²⁴ <https://docs.docker.com/engine/swarm/swarm-tutorial/>

²⁵ <https://docs.docker.com/release-notes/docker-ce/>

²⁶ Image taken from <https://blog.docker.com/2015/11/deploy-manage-cluster-docker-swarm/>

The first concept introduced by the new swarm mode is the concept of a service. The idea is that users should declare services instead of containers. A service could consist of either one or multiple containers (called tasks) and includes definitions such as which networks the containers of that service will connect to, how many replicas of the service there will be, etc.

Swarm also introduces an internal implementation of service reconciliation. This means, if the user declares that he needs 3 tasks of the Orion service, when one task fails, Docker will automatically reschedule a new task to conciliate the required status of 3 tasks.

With an internal DNS in the swarm nodes, there is no need to reference the deployed services containers by their IPs. Being able to reference them by name simplifies the operation of the services in the swarm, particularly considering the dynamic nature of the containers and the fact that, due to redeployments, those IPs might be subject to change.

When having multiple instances of a service, by default a virtual IP is created for the service. Swarm managers use an internal load balancer to distribute requests on that virtual IP among the IP addresses assigned to the single task instances part of a service. This behaviour can be changed if required, as we will explore in further sections for some of the recipes.

Services can be deployed in different *modes*, but there are mainly two options: global or replicated. A global service means Docker will deploy a task for that service on each node of the cluster. The replicated mode instead will let the user define the number of replicas (tasks) for a given service and Docker will spread those replicas among the available swarm nodes as fairly as possible. Below an extract of a Docker Compose file used in the recipe for the Context Broker in HA.

```
...
services:
  orion:
    image: fiware/orion:1.3.0
    ...
    deploy:
      replicas: 3
    networks:
      - mongo-replica_replica
...
```

Moreover, Docker provides two more mechanisms to adjust the deployment strategy when particular requirements should be met, namely *constraints* and *preferences*. Both of these are used in combination with the *labels* mechanisms for swarm nodes. This way, developers first tag the infrastructure nodes with labels as shown below and then refer to them in the rules (constraints or preferences) defined in each service declaration.

```
$ docker node update --label-add fiware.region=region1 node1
$ docker node update --label-add fiware.region=region2 node2
```

These features to control deployment could be helpful for example in the applicability of a multi-site pattern in the deployment of a service, where the location of service replicas should be constrained (or preferred) to a specific set of nodes of a multi-site cluster. The following is an extract example of how a service is declared to be executed only in manager nodes running Ubuntu 14.04 and preferably spread across fiware lab regions (regardless of the number of nodes each region has).

```
...
placement:
  constraints:
    - node.role == manager
    - engine.labels.operatingsystem == ubuntu 14.04
  preferences:
```



```
...
- spread: node.fiware.region
```

This was just a brief exemplification of the Docker concepts SmartSDK builds atop. For more details, readers should refer to the official documentation²⁷.

Nevertheless, it is worth mentioning that though very powerful, Docker technologies are not a silver bullet to attend all the requirements. There are still some challenges and open issues to be tackled such as autoscaling. This is partly due to the fact that Compose and Swarm are relatively new technologies and, even though they have a really fast pace of development, there are still much work in progress, as it can be seen in their Github Issues page²⁸.

2.5.2 Documenting and Developing Architecture Patterns recipes

SmartSDK recipes will be developed mainly in the form of Docker Compose files with supplementary script files when needed. All these files will be kept in a GitHub repository along with the corresponding documentation for developers and users. The repository is already available and can be explored at SmartSDK's GitHub organisation [6]. Moreover, the official documentation is available online [7].

Recipes are organized in different folders, mainly reflecting their relation to different components of the FIWARE Architecture. For example, at the root of the folders structure there is currently one folder for the Data Management chapter, where recipes for Generic Enablers such as Orion are placed. Likewise, there is a different folder called IoT Management intended to host the recipes for the IoT stack generic enablers.

Each folder must contain the following elements:

- ➔ An introduction explaining the purpose of the recipe, explained in a Markdown²⁹ file. For example, which GE or combination of GEs will be used by the recipe.
- ➔ Links to proper pieces of external documentation. For example, instead of explaining details of the Generic Enablers, links to the corresponding FIWARE documentation sites should be provided. It is not a good practice to significantly repeat documentation because it is error prone and makes content more likely to become outdated.
- ➔ A high-level overview diagram explaining the main ideas of the recipe, if not already covered by the previous point. For example, Figure 22 shows the high-level overview for the MongoDB replicaset recipe.

²⁷ <https://docs.docker.com/engine/swarm/services/#control-service-placement>

²⁸ <https://github.com/docker/docker/issues>

²⁹ <https://daringfireball.net/projects/markdown/>

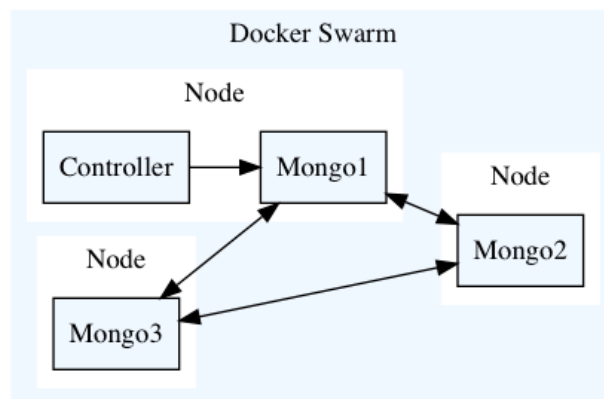


Figure 22: Architecture of the MongoDB replicaset recipe.

- ➔ Instructions on how to use the recipe, and eventually customize it. Since each user might desire different kind of customizations, it may not always be possible to cover all customization options. However, something like a walkthrough showing why default values work as they do will let users understand how to implement the changes according to their needs.
- ➔ The recipe itself and all complementary files required for proper use.

Furthermore, the repository will contain required instructions to create and share either new recipes or enhancements of the existing one. This way, developers can improve the SmartSDK repository by sharing their expertise and tools. Guidelines will also be developed to keep an order in the toolbox. For example, stating which documentation tools to use, standard folders structure and file formats, etc. All of this is covered by a “How to Contribute” section of the documentation.

2.5.3 SmartSDK Architecture Pattern recipes

SmartSDK recipes are designed for different services of the main two FIWARE Architecture Chapters, namely, the Data Management and Internet of Things Services Enablement. A third category groups miscellaneous recipes that may not always be strictly related to FIWARE but that are of great utility for the development of the recipes. This section will briefly explain the first recipes that were developed so far as well as outline some of the foresaw applicable recipes for different generic enablers of the mentioned chapters.

It is worth noting that this is not an exhaustive list and new recipes might be developed throughout the lifespan of the project (and hopefully beyond). Moreover, recipes are and should always be designed in a lean and modular way so that they can be combined to create new ones. For example, reusing the recipe for a MongoDB replicaset, users can create a recipe to use the Orion Context Broker storing its data in that replicaset. Thus, it becomes unfeasible to list in this document all possible combinations of recipes.

2.5.3.1 MongoDB replicaset

The first developed recipe was that of a MongoDB replicaset. The task involved deploying a service of MongoDB instances in the Docker Swarm. In the spirit of the HA pattern, the idea was to have each replicaset member automatically deployed to a different machine (node) of the swarm. This way, if a node failed, it would not take away all the replicas at once. This was achieved using the “global” deploy mode of Docker Compose, which instructs Docker to deploy one container (task) of that service in each available node.

One of the main challenges was automating the configuration of the replicaset. Configuring a replicaset is not so complicated when you have the IP addresses of each member of the replica. But, the problem in a Docker Swarm is that by default you do not know a priori which IPs each container

will be assigned in the overlay network. Moreover, at the time being, Docker does not provide a safe introspective API³⁰ which containers could request for these pieces of information.

To circumvent this issue, a side-car controller script was developed to retrieve these required pieces of information once everything is deployed and connect to a mongo container to configure the replicaset. For the time being, it is using the Docker API of the host daemon, a very popular workaround in the Docker community, though not ideal from a security point of view. Nonetheless, the script can be easily updated once a safer mechanism is provided by Docker. The nitty-gritty of the replicaset configuration remains basically the same. For further details please refer to the recipe documentation files³¹.

Finally, adding nodes to the cluster will imply new instances of the mongo will be added to the cluster, which will be automatically be added to the replicaset by the side-car controller.

For more details, please refer to the official recipe documentation at <https://smartsdk.github.io/smartsdk-recipes/utis/mongo-replicaset/readme/>.

2.5.3.2 Orion Context Broker

The following step was to create a new recipe for the frontend of Orion Context Broker in HA mode that would reuse the MongoDB replica recipe.

Client requests for Orion will arrive at Docker's internal Load Balancer (LB). This component will make sure that requests are equally distributed to all the tasks (containers) of the Orion service. It is not that the same request arrives at all containers at once, rather, request will be dispatched to different containers.

Thanks to Docker's embedded DNS server, the Orion service does not attach to any specific mongod instance. It was linked to the "MongoDB" service, who has a virtual IP, whose traffic is then automatically directed to a final task of that service.

Users could then manually scale up the service running a simple command like "*docker service scale orion=5*" to scale from 3 (in the example above) to 5 instances of the Orion service. Ideally, the scaling action would be automated to become a function of a specific group of metrics. For example, a new replica should be deployed whenever the average response time falls below a certain threshold. However, the support for this kind of orchestration is something still not fully developed in Docker and hence the recipes will have to use operator's defined number of replicas.

For more details, please refer to the official recipe documentation at <https://martel-innovate.github.io/smartsdk-recipes/data-management/context-broker/readme/>.

2.5.3.3 Comet STH

This recipe allows users to deploy scalable instances of Comet³² on top of a Docker cluster. Because Comet uses MongoDB as the database, and due to the fact that the Comet service is stateless, this recipe shows strong resemblance with the one for Orion. This means, for the backend part it reuses the MongoDB recipe, and for the front end it uses the official FIWARE Docker image. The Comet service can be configured through environment variables at the launch time, as it can be seen in the recipe compose file in the recipes git repository.

For more details, please refer to the official recipe documentation at <https://smartsdk.github.io/smartsdk-recipes/data-management/sth/readme/>.

³⁰ <https://github.com/docker/docker/issues/8427>

³¹ <https://martel-innovate.github.io/smartsdk-recipes/utis/mongodb/replica/readme/>

³² <https://fiware-sth-comet.readthedocs.io/en/latest/>

2.5.3.4 Cygnus

This recipe allows user to deploy, by default, a high-available cygnus-ngsi with an instance of mysql as a backend example. The accompanying documentation explains how to adjust the recipe to use different backends.

Moreover, the recipe was prepared to leverage on the usage of Docker config files. This way, the user can simply customise the config files in the recipe folder and redeploy Cygnus (for example, to change the type of channels) without the need to go through the complex process of recreating the docker image of cygnus out of changes in a git repository.

For more details, please refer to the official recipe documentation at <https://smartsdk.github.io/smartsdk-recipes/data-management/cygnus/readme/>.

2.5.3.5 QuantumLeap

This recipe lets user configure their own HA deployment of quantumleap on top of docker swarm clusters. Through well-documented environment variables, the user can specify the nature of the cluster (for example the number of nodes) and the recipe will automatically deploy both frontend and backend of QuantumLeap with as many instances as specified.

The recipe declares a volume to persist data by default (in case containers are restarted) and also offers a second deployment file used to deploy complementary services such as grafana, which would be automatically configured to integrate the required plugin to work with QuantumLeap's backend.

For more details, please refer to the official recipe documentation at <https://smartsdk.github.io/smartsdk-recipes/data-management/quantumleap/readme/>.

2.5.3.6 IDAS

This is a group of recipes, that allow users to deploy scalable instances of the most used versions of FIWARE IoT Agents (ul, json and lwm2m) ³³.

This recipe also makes heavy use of docker configs feature, which lets users simplify the configuration of both the agent as well as the MQTT broker (when used). This way, users can change a single file for the configuration and the new configuration will be applied automatically to as many replicas as desired.

For more details, please refer to the official recipe documentation at <https://smartsdk.github.io/smartsdk-recipes/iot-services/readme/>.

2.5.3.7 API Umbrella

This recipe allows user to deploy the core elements of the API Umbrella³⁴ solution for API Management. It reuses the MongoDB recipe to deploy the backend database where some data are stored. It also leverages on Docker's config files support to simplify user configuration of the service and also scalability of such instance. As documented in the recipe, at the moment and until more recent versions of the complementary services are supported, this recipe is not covering those.

For more details, please refer to the official recipe documentation at <https://github.com/smartsdk/smartsdk-recipes/tree/master/recipes/security/api-umbrella>.

³³ <https://catalogue.fiware.org/enablers/backend-device-management-idas/documentation>

³⁴ <https://apiumbrella.io/>

2.5.4 Roadmap for SmartSDK Recipes

The roadmap for the SmartSDK recipes remains at this point in time to a great extent like the one presented in the previous version of this deliverable. The only noticeable changes include the delay of the CKAN and its Open Data Integration recipe and the introduction of two additional new recipes.

Since CKAN is not used for now in any of the application scenarios, and most of the times is not self-deployed in applications, we decided to move it to the end of list with lowest priority, to be developed if time allows. On the other hand, we introduced two recipes that were not foreseen in the original plan: One for QuantumLeap and one for API Umbrella. The former is a component incubated in this same project, already being used in the applications so it was natural to develop recipes for it. The later cover aspects of API management and security, two fundamental aspects for new applications.

As a reminder, the SmartSDK project is being tracked in JIRA³⁵ and all the tasks related to the development of the recipes are identified by the JIRA component “Generic Architecture Patterns”.

2.5.4.1 Testing recipes and their scalability

To understand the behaviour of the proposed reference architectures and their implementation into recipes, it is important to test the deployed recipes with proper workloads. To this aim, we developed a tool: the NGSI Load Tester³⁶. The tool will be adopted in the next future to test the recipes and measure their behaviour with the increase of load and in case of service failures.

³⁵ <https://jira.fiware.org/projects/SMART>

³⁶ <https://github.com/smartsdk/ngsi-load-tester>

3 SMARTSDK DATA MODELS

The FIWARE Community is promoting the development of re-usable and harmonized data models under the umbrella of the FIWARE data models³⁷ initiative. These data models are reusing and extending the work performed under the GSMA IoT Big Data initiative. The work conducted under FIWARE is evolving on a daily basis by taking into consideration requirements from adopters of data models. All FIWARE data models, coherently with the centric role of NGSIv2 API, are expressed to be used with such API.

SmartSDK contributes to this activity in different ways:

- ➔ Developing Applications that reuse existing FIWARE data models;
- ➔ Providing adapters to convert existing data into FIWARE data models;
- ➔ Populating the Global FIWARE Context Broker with data compliant with existing FIWARE data models;
- ➔ Proposing extensions to existing FIWARE data models according to the SmartSDK applications' requirements;
- ➔ Developing novel FIWARE data models according to the SmartSDK applications' requirements;
- ➔ Providing tools that support the FIWARE data models activity;

In the 18 months of activities, SmartSDK contributed significantly to FIWARE data models as documented also in D2.1. In the following section we summarise existing FIWARE data models, and contributions provided so far by SmartSDK and their status in the community.

3.1 Overview of existing FIWARE data models

As early mentioned, the FIWARE data models initiative is steering the development of harmonised data models to provide a set of common models within the FIWARE community to enable data portability at the application layer.

Adoption of common models, beside the adoption of common APIs (NGSIv2 in the case of FIWARE), is a fundamental step to allow for a scalable data ecosystem that supports the interoperability and re-usage of data and applications working on top of data.

In line with this principle, FIWARE has harmonised so far the following set of data models:

- ➔ **Environment.** A model to enable the monitoring of air quality and other environmental conditions for a healthier living. In particular, covered entity types include:
 - AirQualityObserved: an observation of air quality conditions at a certain place and time.
 - WaterQualityObserved: capture all the parameters involved in Water Quality scenarios.
 - NoiseLevelObserved: represents an observation of those parameters that estimate noise pressure levels at a certain place and time.
- ➔ **Civic Issue tracking.** This set provides entity types for civic issue tracking interoperable with the de-facto standard Open311. In particular, covered entity types include:
 - ServiceType. A type of service a citizen can request.

³⁷ schema.fiware.org

- ServiceRequest. A specific service request (of a service type) made by a citizen.
- ➔ **Street Lighting.** It models street lights and all their controlling equipment towards energy-efficient and effective urban illuminance. The covered entity types include:
 - Streetlight: a particular instance of a streetlight. A streetlight is composed by a lantern and a lamp. Such elements are mounted on a column (pole), wall or other structure.
 - StreetlightGroup: a group of streetlights being part of the same circuit and controlled together by an automated system.
 - StreetlightModel: a model of streetlight composed by a specific supporting structure model, a lantern model and a lamp model. A streetlight instance will be based on a certain streetlight model.
 - StreetlightControlCabinet: an automated equipment, usually on street, typically used to control a group or groups of streetlights, i.e. one or more circuits.
- ➔ **Device.** This set of entity types describes IoT devices (sensors, actuators, wearables, etc.) with their characteristics and dynamic status. The covered entity types include:
 - Device: an electronic apparatus designed to accomplish a particular task.
 - DeviceModel: the static properties common to multiple instances of a Device.
- ➔ **Transportation.** Transportation data models for smart mobility and efficient management of municipal services. The covered entity types include:
 - TrafficFlowObserved: a recorded observation of traffic flow.
 - Road: a geographic and contextual description of a Road.
 - RoadSegment: a geographic and contextual description of a road segment.
 - Vehicle: a specific vehicle instance.
 - VehicleModel: a model of vehicle, capturing its static properties such as dimensions, materials or features.
- ➔ **Indicators.** It models key performance indicators intended to measure the success of an organization or of a certain activity in which the organisation is engaged.
- ➔ **Waste Management.** This model enable efficient, recycling friendly, municipal or industrial waste management using containers, litters, etc. The covered entity types include:
 - WasteContainerIsle: the isle that holds one or more containers.
 - WasteContainerModel: a model of waste container, capturing its static properties such as dimensions, materials or features.
 - WasteContainer: a certain instance of waste container placed at a particular isle or place.
- ➔ **Parking.** This model provides real time and static parking data (on street and off street) interoperable with the EU standard DATEX II. This model includes the following entity types:
 - OffStreetParking: an offstreet parking site with explicit entries and exits.
 - ParkingAccess: an access point to an off-street parking site.
 - OnStreetParking: an on street, free entry (but might be metered) parking zone which contains at least one or more adjacent parking spots.
 - ParkingGroup: a group of parking spots.
 - ParkingSpot: an individual, usually monitored, parking spot.

- ➔ **Weather.** Weather observed, weather forecasted or warnings about potential extreme weather conditions.

The FIWARE data models have been adopted in different applications scenarios enabling their validation, for example in different cities.

SmartSDK is developing demonstrators in three domains:

- ➔ **Smart Cities:** the planned demonstrator will provide an application that helps citizen to move around their city in environment-friendly and healthy way.
- ➔ **Smart Security:** the planned demonstrator monitor parking spots and buildings for security threats.
- ➔ **Smart Health:** the planned demonstrator collect and analyse patient behaviour data through mobile devices.

Clearly, several of the data models already part of FIWARE data models initiative are key to SmartSDK. Table 1: FIWARE data models by Scenario summarises a mapping between scenarios and FIWARE data models.

FIWARE Data Model	Scenario	Comments
Transportation	Smart City	The model will be used for tracking public and private vehicles to support computation of eco-friendly and health-friendly paths. SmartSDK extended the model to provide support for public transport routes and related services.
Weather	Smart City	The model will be used for the weather forecast to support computation of eco-friendly and health-friendly paths.
Environment	Smart City	The model will be used for air quality data to support computation of eco-friendly and health-friendly paths. Several fixes have been provided to the data model by SmartSDK, including the official JSON Schema
Device	Smart Security	The model will be used to define and track the security devices deployed in the building and parkings to monitor security. Several fixes have been provided to the data model by SmartSDK, including the official JSON Schema.
Transportation	Smart Security	The model will be used to track vehicles detected by camera and sensors.
Parking	Smart Security	The model will be used to support the tracking of vehicles in parking lots.

Table 1: FIWARE data models by Scenario

3.2 Documenting and Developing Data Models

The existing FIWARE data models are documented using markdown format as detailed in the data model template³⁸, and they are formalised using JSON Schema draft version 4 [8]. The data model

³⁸ The data model template is available at: https://github.com/Fiware/dataModels/blob/master/datamodel_template.md

documentation must include:

- ➔ A general description of the data model.
- ➔ A description of the attributes of the data model that are formalised in the JSON Schema.
- ➔ An example of data model instance as JSON code fragment.
- ➔ A description of reference endpoints that store data following the specified data model.
- ➔ A list of activities yet to be completed on the data model.

The data model template is complemented by a set of basic data model references that are often reused in the different data models. These include:

- ➔ Common general properties (e.g. name, data source, ...) inherited from the GSMA initiative [9].
- ➔ Common properties to define location (e.g. address, position, ...) based on GeoJSON.

There is a current ongoing discussion on how to better support the navigation of data model documentation. The plan is to provide FIWARE data models descriptions through a portal that follows the principles of the schema.org³⁹ initiative.

Besides providing a template for the data model development, FIWARE defined a set of guidelines and related design principles that should be followed to develop a FIWARE Data Model⁴⁰:

- ➔ **Syntax guidelines.** They define naming conventions for entity types or attributes. For instance, the camelcase⁴¹ naming pattern is recommended.
- ➔ **Reusing guidelines.** They promote reusability encouraging the authors of new data models to check existing work (eg. Openstreetmap or schema.org) and base on it when feasible.
- ➔ **Modelling guidelines.** They provide modelling design patterns to be followed when representing real world entities and attributes. They define the recommended practices on modelling dates, location, civic addresses, etc. or how to express extra attribute metadata, for instance, timestamps.
- ➔ **Representation guidelines.** They recommend, particularly, how to model and represent quantitative values. Percentage-based magnitudes, such as relative humidity must be represented in parts per one. An important aspect of a quantitative value is the unit of measurement. To this aim, the guidelines advice on the usage of the International System of Units [10] by default.

To contribute to the existing Data Model repository on Github⁴², developers are recommended to fork the repository in Github, add the new models or revise the existing ones and provide your contributions as a pull request to the original Github repository.

3.2.1 Automated support to FIWARE Data Model validation

Following the requirement by the FIWARE community to facilitate the development of models by having easy means to validate them, and simplify their maintenance, SmartSDK developed a solution to validate FIWARE data models that is currently integrated in the Continuous Integration (CI) workflow defined for FIWARE data models. Code of the FIWARE Data Model validator is available in FIWARE Data Model github project: <https://github.com/Fiware/dataModels/tree/master/validator>

³⁹ <https://schema.org>

⁴⁰ The on going work on recommendations is described at <http://fiware-datamodels.readthedocs.io/en/latest/guidelines/index.html>

⁴¹ https://en.wikipedia.org/wiki/Camel_case

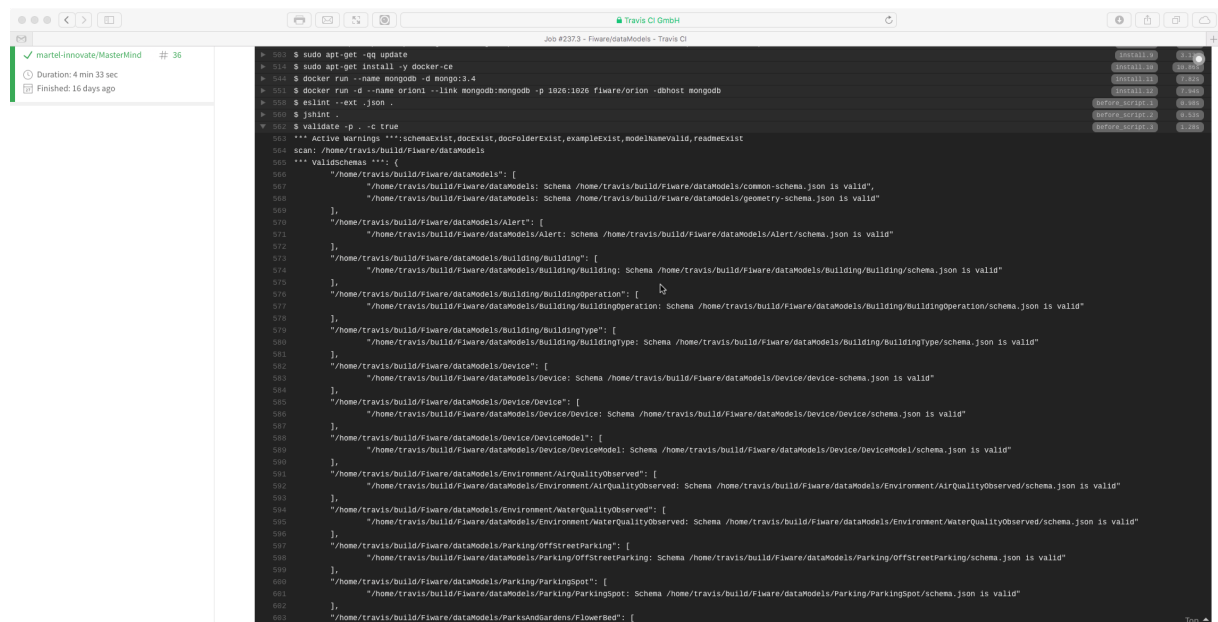
⁴² <https://github.com/Fiware/dataModels>

FIWARE Data Model validator is an utility to help the management of NGSI Data Models. Its code leverages on the AJV JSON Schema Validator⁴³.

The FIWARE Data Model validator perform the following checks for each Data Model:

- ➔ validity of JSON Schema
- ➔ validity of JSON Examples
- ➔ support of JSON Examples in Orion Context Broker
- ➔ adherence of Data Model name to FIWARE Data Models guidelines
- ➔ existence of Doc folder
- ➔ existence of JSON Schema
- ➔ existence of one or more JSON examples
- ➔ existence of README file

Figure 23 shows the results of a recent validation⁴⁴ run as part of the FIWARE data model CI workflow.



```

$ sudo apt-get -qq update
$ sudo apt-get install -y docker-ce
$ docker run --name mongod -d mongo:3.4
$ docker run -d --name orion --link mongod:mongod -p 1026:1026 fiware/orion -dhost mongod
$ eslint --ext .json .
$ jshint
$ validate -p -c true

*** Active Warnings ***: schematExist, docExist, docFolderExist, exampleExist, modelNameValid, readmeExist
scan: /home/travis/build/Fiware/dataModels
*** ValidSchemas ***:
  "/home/travis/build/Fiware/dataModels": [
    "/home/travis/build/Fiware/dataModels: Schema /home/travis/build/Fiware/dataModels/common-schema.json is valid",
    "/home/travis/build/Fiware/dataModels: Schema /home/travis/build/Fiware/dataModels/geometry-schema.json is valid"
  ],
  "/home/travis/build/Fiware/dataModels/Alert": [
    "/home/travis/build/Fiware/dataModels/Alert: Schema /home/travis/build/Fiware/dataModels/Alert/schema.json is valid"
  ],
  "/home/travis/build/Fiware/dataModels/Building/Building": [
    "/home/travis/build/Fiware/dataModels/Building/Building: Schema /home/travis/build/Fiware/dataModels/Building/Building/schema.json is valid"
  ],
  "/home/travis/build/Fiware/dataModels/Building/BuildingOperation": [
    "/home/travis/build/Fiware/dataModels/Building/BuildingOperation: Schema /home/travis/build/Fiware/dataModels/Building/BuildingOperation/schema.json is valid"
  ],
  "/home/travis/build/Fiware/dataModels/Building/BuildingType": [
    "/home/travis/build/Fiware/dataModels/Building/BuildingType: Schema /home/travis/build/Fiware/dataModels/Building/BuildingType/schema.json is valid"
  ],
  "/home/travis/build/Fiware/dataModels/Device": [
    "/home/travis/build/Fiware/dataModels/Device: Schema /home/travis/build/Fiware/dataModels/Device/device-schema.json is valid"
  ],
  "/home/travis/build/Fiware/dataModels/Device/Device": [
    "/home/travis/build/Fiware/dataModels/Device/Device: Schema /home/travis/build/Fiware/dataModels/Device/Device/schema.json is valid"
  ],
  "/home/travis/build/Fiware/dataModels/Device/DeviceModel": [
    "/home/travis/build/Fiware/dataModels/Device/DeviceModel: Schema /home/travis/build/Fiware/dataModels/Device/DeviceModel/schema.json is valid"
  ],
  "/home/travis/build/Fiware/dataModels/Environment/AirQualityObserved": [
    "/home/travis/build/Fiware/dataModels/Environment/AirQualityObserved: Schema /home/travis/build/Fiware/dataModels/Environment/AirQualityObserved/schema.json is valid"
  ],
  "/home/travis/build/Fiware/dataModels/Environment/WaterQualityObserved": [
    "/home/travis/build/Fiware/dataModels/Environment/WaterQualityObserved: Schema /home/travis/build/Fiware/dataModels/Environment/WaterQualityObserved/schema.json is valid"
  ],
  "/home/travis/build/Fiware/dataModels/Parking/OffStreetParking": [
    "/home/travis/build/Fiware/dataModels/Parking/OffStreetParking: Schema /home/travis/build/Fiware/dataModels/Parking/OffStreetParking/schema.json is valid"
  ],
  "/home/travis/build/Fiware/dataModels/Parking/ParkingSpot": [
    "/home/travis/build/Fiware/dataModels/Parking/ParkingSpot: Schema /home/travis/build/Fiware/dataModels/Parking/ParkingSpot/schema.json is valid"
  ],
  "/home/travis/build/Fiware/dataModels/ParksAndGardens/FlowerBed": [

```

Figure 23. Example of outcomes of FIWARE Data Model validator.

3.2.1.1 Install the validator

To install the validator on your machine, you need nodejs 7.0.0+. Instructions on how to install nodejs are available here: <https://nodejs.org/en/download/package-manager/>.

Once nodejs installed in your system, you can install the validator with the following command:

```
npm install -g fiware-model-validator
```

⁴³ <https://github.com/epoberezkin/ajv>

⁴⁴ <https://travis-ci.org/Fiware/dataModels/jobs/345181698>

3.2.1.2 Using the validator

To use the validator, execute it from the root of the DataModel repository:

```
validate -p DataModel -w ignore -i [common-schema.json,geometry-
schema.json]
```

Command line available options are:

- ➔ `-i, --dmv:importSchemas`. Additional schemas that will be included during validation. Default imported schemas are: `common-schema.json`, `geometry-schema.json`
- ➔ `-w, --dmv:warnings`. How to handle FIWARE Data Models checks warnings.
 - `true` (default) - print warnings, but does not fail.
 - `ignore` - do nothing and do not print warnings.
 - `fail` - print warnings, and fails.
- ➔ `-p, --dmv:path`. The path of FIWARE Data Model(s) to be validated (if recursion enabled, it will be the starting point of recursion)
- ➔ `-c, --dmv:contextBroker`. Enable example testing with Orion Context Broker
- ➔ `-u, --dmv:contextBrokerUrl`. Orion Context Broker URL for Example testing
- ➔ `-v, --version`. Print the validator version
- ➔ `-h, --help`. Print the help message

For a more fine grained configuration you can create a `config.json` file. An example is provided in the repository.

3.3 Novel SmartSDK Data Models

In addition to the models identified in Section 3.1, SmartSDK scenarios require additional data models that are not yet covered by FIWARE data models initiative. The first release of such models was completed in D2.1 at M9 and will be further refined in D2.4 (planned for M21). Some of the discussed models have been already included in the official FIWARE Data Model repository.

In this deliverable, for each scenario we provide a short discussion on contributed development so far. Complete model description can be found in D2.1 or in FIWARE Data Model repository.

3.3.1 Smart City

The Smart City scenario, besides reusing the harmonized transportation models developed by FIWARE, requires models for the **Transportation Schedule**⁴⁵, **Alert**, **PollenLevelObserved**, **Smart POI** and **Smart Spot**. The Transportation Schedule data model is required to support the route planning based on public transports routes. The model will be based on the standard de facto General Transit Feed Specification (GTFS) model. The Alert⁴⁶ data model aims at supporting scenarios in which a smart platform sends alerts related to traffic jam, accidents, weather conditions, high level of pollutants and so on. The purpose of the model is to support the generation of notifications for a user or trigger other actions, based on such alerts. The PollenLevelObserved model allows to model the

⁴⁵ <https://developers.google.com/transit/gtfs/>

⁴⁶ <https://github.com/Fiware/dataModels/tree/master/Alert>

current quantity and allergen level of pollens within a City. The Smart POI defines an interactive point which provides information, entertainment or co-creation tools to citizens. Optionally it can reference a related smart city asset with enriched interaction provided by this technology. The Smart Spot defines a set of resources related to a physical device and the technology to provide a Smart Point of Interaction.

3.3.2 Smart Security

The Smart City scenario, besides reusing the harmonized for parking, devices and transportation developed in FIWARE, requires models to support the description of **Building**, **VideoObject**, **VisualObjects** and **User Context**. The Building data model is used to describe surveilled buildings. It is based on the Building model proposed by GSMA IoT Big Data Harmonised Data Model initiative. The VideoObject data model allows the storage of metadata about recorded surveillance video and to annotated them with in respect detected security events. The model is based on relevant video metadata models. The VisualObject is used to describe identified objects in a video streaming. The User Context⁴⁷ data model allows to describe the context of a given (anonymised) user, e.g. the activity he is currently performing or his current location.

3.3.3 Smart Health

The Smart Health scenario, focusing on acquisition of health parameters from mobile devices, covers topics not yet covered in the FIWARE Data Model initiative, neither by previous FIWARE projects in the healthcare area (FI-STAR). The work conducted in the scenario builds on top of the Open Mobile Health⁴⁸ standard. Entities modelled include: the **Physical Test**, the **Questionnaire**, and the **Control Test**. The Physical Test collect information from sensor in relation to Patients; the Questionnaire entity contains information collected by doctors after a Physical Test; and the Control Test collect several patient measurements to be related with the Physical Test.

⁴⁷ <https://github.com/Fiware/dataModels/tree/master/User>

⁴⁸ <http://www.openmhealth.org>

4 CONCLUSIONS AND FUTURE WORK

This deliverable outlines one of the main contributions of SmartSDK to the FIWARE Community and to the reinforcement of the collaboration between Europe and Mexico around FIWARE. The ability to replicate, in an easy way, the EU experience in Mexico and vice-versa relies on defining reference implementations for a given application scenario. A reference implementation for a Smart application built using FIWARE components relies on: reference components, reference data models and reference architectures to glue everything together.

SmartSDK, with the aim of facilitating FIWARE adoption beyond simple proof of concepts, invests in the development of reference architectures that, benefitting from Cloud Architecture Patterns, provide platform deployments that deals with production grade requirements such as fault tolerance and ability to scale.

Compared the previous version, this final version of the deliverable includes several advancements. As regards tooling the advancements include:

- A library of reusable architecture pattern to easily create diagrams.
- A validation tool to support the creation of new data models and their continuous integration.

Within respect recipes, we know support the following additional FIWARE services:

- API Umbrella to secure APIs.
- Cygnus, Comet STH, QuantumLeap.
- IoT IDAS agents (ul, json and lwm2m).

Finally, as regards data models, the following data models have been further developed and proposed to the FIWARE Community (some of them have been already validated and included):

- Alert data model that supports scenarios in which a smart platform sends alerts related to traffic jam, accidents, weather conditions, high level of pollutants and so on.
- PollenLevelObserved data model that models the current quantity and allergen level of pollens.
- Smart POI data model defines an interactive point which provides information, entertainment or co-creation tools to citizens.
- Smart Spot data model defines a set of resources related to a physical device and the technology to provide a Smart Point of Interaction.
- User Context data model allows the description of the context of a given (anonymised) user, e.g. the activity he is currently performing or his current location.

We believe that these activities will be a corner stone in the future evolution of FIWARE and we are happy to know that some of the results have been already adopted in other projects.

Beyond the reported activities in this deliverable, in the next future, according to evolutions in WP2 and WP3, additional data models and recipes may be developed and included in the online repositories.

REFERENCES

- [1] FIWARE NGSIv2: <http://telefonicaid.github.io/fiware-orion/api/v2/stable/>
- [2] Hortonworks Data Platform. (Dec 21, 2015). Hadoop High Availability. https://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.3.4/bk_hadoop-ha/bk_hadoop-ha-20151221.pdf
- [3] Gamma, E., Helm, R., Johnson, R. E., & Vlissides, J. (2016). *Design patterns: elements of reusable object-oriented software*. Boston, MA: Addison-Wesley.
- [4] Wilder, B. (2012). *Cloud architecture patterns*. Beijing: O'Reilly
- [5] Eric Brewer, “CAP twelve years later: How the "rules" have changed”, Computer, Volume 45, Issue 2 (2012), pg. 23–29.
- [6] SmartSDK Github Repository: <https://github.com/smartsdk/smartsdk-recipes>
- [7] SmartSDK Documentation: <https://smartsdk.github.io/smartsdk-recipes/>
- [8] JSON Schema draft version 4: <https://tools.ietf.org/html/draft-zyp-json-schema-04>
- [9] GeoJSON format: <https://tools.ietf.org/html/rfc7946>
- [10] Taylor, B. N. (n.d.). The International System of Units (SI) (National Institute of Standards and Technology (U.S.)). Retrieved from <http://physics.nist.gov/Pubs/SP330/sp330.pdf>
- [11] IoT Big Data Harmonised Data Model(Publication). (2016, October 25). Retrieved <http://www.gsma.com/connectedliving/wp-content/uploads/2016/11/CLP.26-v1.0.pdf>
- [12] SmartSDK Consortium. Description of Action. July 2016.