Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Trabajo Fin de Máster

A Proposal for a Business Framework targeted at Emerging Business Services and Applications

Author: Francisco de la Vega García

Director: Francisco Javier Soriano Camino

Resumen

Un *Service Business Framework* consiste en una serie de componentes interrelacionados que permiten la gestión de servicios de negocio a través de su ciclo de vida, desde su creación, descubrimiento y comparación, hasta su monetización (incluyendo un posible reparto de beneficios). De esta manera, el denominado *FIWARE Business Framework* trata de permitir a los usuarios de la plataforma FIWARE mejorar sus productos con funcionalidades de búsqueda, describrimiento, comparación, monetización y reparto de beneficios. Para lograr este objetivo, el Business Framework de FIWARE proporciona la especificación abierta y las APIs de una serie de componentes (denominados "Generic Enablers" en terminología FIWARE), junto con una implementación de referencia de las mismas pueden ser facilmente integradas en los sitemas existentes para conseguir aplicaciones con valor añadido.

Al comienzo de este trabajo de fin de master, el Business Framework de FIWARE no era lo suficientemente maduro como para cubrir los requisitos de sus usuarios, ya que ofrecía modelos demasiado generales y dejaba algunas funcionalidades clave para ser implementadas por los usuarios. Para solucionar estos problemas, el principal objectivo desarrollado en el contexto de este trabajo de fin de master ha consistido en mejorar y evolucionar el Business Framework de FIWARE para dar respuesta a las demandas de sus usuarios.

Para alcanzar el pricipal objetivo propuesto, el Business Framework de FIWARE ha sido evaluado usando la información proporcionada por los usuarios de la plataforma, principalmente PyMEs y start-ups que usan este framework en sus soluciones, con el objetivo de obtener una lista de requisitos y de diseñar a partir de éstos un roadmap de evolución a 6 meses. Después, los differentes problemas identificados se han tratado uno por uno dando en cada caso una solución capaz de cubrir los requisitos de los usuarios. Finalmente, se han evaluado los resultados obtenidos en el proyecto integrando el Business Framework desarrollado con un sistema existente para la gestión de datos de consusmo energético, construyendo lo que se ha denominado Mercado de Datos de Consumo Energético. Esto además ha permitido demostrar la utilidad del framework propuesto para evolucionar una plataforma de datos abiertos bien conocida como es CKAN a un verdadero mercado de datos.

Abstract

Service Business Frameworks consist on a number of interrelated components that support the management of business services across their whole lifecycle, from their creation, publication, discovery and comparison, to their monetization (possibly including revenue settlement and sharing). In this regard, the FIWARE Business Framework aims at allowing FIWARE users to enhance their solutions with search, discovery, comparison, monetization and revenue settlement and sharing features. To achieve this objective, the FIWARE Business Framework provides the open specification and APIs of a comprehensive set of components (called Generic Enablers in FIWARE terminology), along with a reference implementation of these APIs,, that can be easily integrated with existing systems in order to create value added applications.

At the beginning of the current Master's Thesis, the FIWARE Business Framework was not mature enough to cover the requirements of the its users, since it provided too general models and leaved some key functionality to be implemented by those users. To deal with these issues, the main objective carried out in the context of this Master's Thesis have been enhancing and evolving the FIWARE Business Framework to accomplish with the demands of its users.

For achieving the main objective of this Master's Thesis, the FWARE Business Framework has been evaluated using the feedback provided by FIWARE users, mainly SMEs and start-ups, actually using the framework in their solutions, in order to determine a list of requirements and to design a roadmap for the evolution and improvement of the existing framework in the next 6 months. Then, the different issues detected have been tackle one by one enhancing them, and trying to give a solution able to cover users requirements. Finally, the results of the project have been evaluated by integrating the evolved FIWARE Business Framework with an existing system in charge of the management of energy consumption data, building what has been called the *Energy Consumption Data Market*. This has also allowed demonstrating the usefulness of the proposed business framework to evolve CKAN, a renowned open data platform, into an actual, fully-fledged data market.

Contents

1 Introduction **1**
1.1 Context of the project 1
1.1.1 The FIWARE platform and the FICORE project 1
1.1.2 The Technical Chapter 1.6: Applications/Services and Data Delivery 3
1.1.3 Previous state of the Business Framework 4
1.2 Motivation of the project 7
1.3 Objectives of the project 8
1.4 Structure of the Rest of the Document 9

2 State of the Art **11**
2.1 Commercial Solutions 11
2.1.1 Consumer Centric Apps Stores 11
2.1.2 Enterprise Store 16
2.1.3 Cloud Services Store 17
2.2 Service Description Languages 19
2.2.1 Technical Description Languages 19
2.2.2 Business-level Description Languages: Linked USDL 22
2.3 Involved technology 24
2.3.1 RDF 24
2.3.2 Open Link Virtuoso 26
2.3.3 OAuth2 27

3 Business Framework Architecture **29**
3.1 Architecture Overview 29
3.2 Identity Management and Access Control 37
3.3 Components Architecture 38
3.3.1 Store Architecture 39
3.3.2 RSS Architecture 48
3.3.3 Marketplace Architecture 52
3.3.4 Repository Architecture 55

4 Offering Model **59**
4.1 Model overview 59
4.2 Pricing Basis 65

4.2.1 Price Plans 66
4.2.2 Pricing Models 66

5 Support for Multiple Digital Assets Support: The Plug-in model 75
5.1 Plug-in Model 76
5.2 Creating a Plug-in 82

6 Pay-Per-Use: The Accounting Proxy 87
6.1 Store Pay-Per-Use Management 87
6.2 Accounting Proxy 90
6.2.1 Accounting Proxy Architecture 91
6.2.2 Integration with the Framework 96

7 Revenue Settlement and Sharing 99
7.1 Revenue Sharing Models 100
7.2 Integration with the Store 103

8 Smart Searches: The Marketplace and The Repository 105
8.1 Repository RDF Support 106
8.2 Marketplace Support for Searches 107

9 Use Case: Energy Consumption Data Market 111
9.1 Overview 111
9.2 Data Market Architecture 113

10 Results and Conclusions 119
10.1 Results 119
10.2 Conclusions 122
10.3 Future lines 123

A Fundamental Modeling Concepts Format 125

B Third Party Components 129
B.1 Identity Manager GE: KeyRock 129
B.2 Context Broker GE: Orion 130
B.3 CKAN 131

List of Figures

2.1 Linked USDL Core . . . 23
2.2 RDF Graph representing Eric Miller . . . 25
2.3 OAuth2 Protocol Flow . . . 28

3.1 FIWARE Business Framework Components Relationships . . . 30
3.2 FIWARE Business Framework Architecture . . . 32
3.3 Offering creation process . . . 35
3.4 Offering acquisition and charging process . . . 36
3.5 Business Framework Authorization Process . . . 38
3.6 Store Architecture . . . 41
3.7 Store API Offering module . . . 43
3.8 Store API Contracting and Search modules . . . 44
3.9 Offering Life Cycle Provider Viewpoint . . . 45
3.10 Offering Life Cycle Customer Viewpoint . . . 45
3.11 RSS Architecture . . . 51
3.12 RSS API Revenue Sharing module . . . 52
3.13 RSS API Expenditure Limits module . . . 52
3.14 Marketplace Architecture . . . 54
3.15 Marketplace API . . . 55
3.16 Repository Architecture . . . 57
3.17 Repository API . . . 57

4.1 Resource USDL model . . . 60
4.2 Offering USDL model . . . 62
4.3 Basic Pricing USDL model . . . 63
4.4 Price Function USDL model . . . 65

5.1 Plug-in Model Architecture . . . 77
5.2 Form for providing a resource . . . 83
5.3 Form for providing basic resource info . . . 84
5.4 Autogenerated form for retrieving resource meta info . . . 86

6.1 Basic Accounting Model . . . 88
6.2 Monitored Services URL Management . . . 91
6.3 Accounting Proxy Architecture . . . 92

6.4 Service access process through the Accounting Proxy 97

7.1 RSS Revenue Sharing Module Architecture 102
7.2 RSS RDF Revenue Sharing model 103

8.1 Repository Integration with a Triple Store 107

9.1 Energy Consumption System Architecture 113
9.2 Energy Consumption Data Market Architecture 116

A.1 FMC Active Componets 126
A.2 FMC Passive Componets 126
A.3 FMC Read Access 126
A.4 FMC Write Access 127
A.5 FMC Read-Write Access 127
A.6 FMC Request - Response 127

List of Tables

2.2 Consumer Centric App Stores 12
2.4 App Stores Models Comparison 15
2.6 App Stores Offering Comparison 15

Chapter 1

Introduction

1.1 Context of the project

1.1.1 The FIWARE platform and the FICORE project

The FIWARE Platform [11] is being developed as part of a project partially funded by the seventh Framework Program of the European Commission, called FI-WARE. This Platform aims to be the core platform of the future internet and its main objective is to increase the global competitiveness of European ICT economy by introducing an innovative infrastructure for cost-effective creation and delivery of versatile digital services, providing high QoS and security guarantees. In this way, the FIWARE Platform is intended for stimulating and cultivating a sustainable ecosystem for both, innovative service providers delivering new applications and solutions in emerging business areas, and end users actively participating in content and services consumption and creation.

The FIWARE Platform is an open platform, based on generic elements (hereunder called *Generic Enablers* or *GEs*) that offer shared and reusable features, useful in multiple usage areas thought various sectors. Note that not all features that are common to applications in a given area can be considered a GE, since is the ability to serve in multiple usage areas that distinguish GEs from what are called *Specific Enablers* (*SEs*), the later being enablers that offer common functionality to applications in a very specific usage area. To clarify this concept, a context broker, a complex event processing system (CEP) or an identity manager can be considered GEs, while a tool for the edition of 3D scenes can be considered an SE. The FIWARE Platform offers a catalogue [9] of already implemented GEs (called *Generic Enabler Reference implementation* or *GEri*) that can be directly used by service providers and entrepreneurs. A service provider, entrepreneur or developer in general can decide to create its own implementation of a GE as long as she adheres to its open specification.

To achieve its objectives, the FIWARE platform focuses on helping innovative

service providers and entrepreneurs easing the creation of context aware applications, hosted in a cloud environment, connected to the internet of things, and able to analyse data at a large scale by providing an easy and innovative way of managing applications and services, that makes processes simple and cost effective, thanks to innovative technological resources and already implemented APIs. In this way, innovative service providers and entrepreneurs can quickly access the market maximizing their opportunities to survive.

Moreover, the FIWARE platform offers some resources to help FIWARE developers and entrepreneurs to carry on with their projects and applications. On the one hand, the FIWARE Academy [8] offers web-casts, tutorials, and training materials on how to use the different components of the platform. On the other hand, the FIWARE Platform includes a free experimentation environment called the FIWARE Lab [10] where all FIWARE resources are ready to be used, so FIWARE developers can find the means to create their projects and test their applications with real data and real users. In addition, more FIWARE environments or nodes apart from the FIWARE Lab can also be created and operated using a set of tools called FIWARE Ops, which supports the worldwide expansion of FIWARE thought the different FIWARE providers.

FICORE is the the FI-WARE continuation project in charge of the evolution of the different GEs and GEris of the FIWARE Platform, the tools of FIWARE Ops, and the maintenance of the FIWARE Lab and the FIWARE Academy, and last but not least, the creation of a real, successful open source community of developers around the FIWARE platform. This project is included in the *Future Internet Public-Private Partnership* (FI-PPP) [7] where it coexists with a set of *Use Case projects* and, more recently, with a number of incubators aimed at supporting Small and Medium Enterprises (SMEs) and entrepreneurs in using FIWARE.

The different Generic Enablers developed within the FICORE project are organized in the following technical chapters:

- **Cloud Hosting**: This chapter provides support for advanced Infrastructure-as-a-service (IaaS) covering the provisioning and management of different physical and virtual resources such as server, block storage, network, and object storage.

- **Data/Media Context Management**: This chapter provides functions to gather, store and publish: (1) discrete data (relevant to defined entities) coming from heterogeneous sources, including sensors, geolocalization sources, external applications and users. (2) Media streams coming from heterogeneous sources, including cameras and external applications. Moreover, this chpater provides the capacity to manage and publish Open Data from internal or external data sources and to use this Open Data to feed or enrich the rest of the GEs of the FIWARE Platform.

- **IoT Services Enablement**: This chapter provides the bridge between the different components within the FIWARE platform and the sensors and actuators that built up the Internet of Things.
- **Advanced Web-Based UI**: This chapter provides a novel approach to advanced interactive user experience for Web applications via an extended version of HTML-5 that adds various new capabilities related to real-time 3D visualization and interaction.
- **Applications/Services and Data Delivery**: This chapter provides the infrastructure for selling, monetizing, and consuming FI applications and services using different pricing models. Additionally, this chapter provides a set of tools for the creation of cockpits and highly configurable dashboards for the visualization of services and data. All the work done in this thesis is in the context of this chapter.
- **Security**: This chapter provides the needed tools to ensure that the acquisition and use of the different components within the FIWARE platform is done according to the existing requirements of security and privacy.
- **Advanced Middleware, Interfaces to Networks and Robotics**: This chapter focuses on a set of differentiated tasks: (1) Provides features for the dynamic monitoring, provisioning and configuration of cloud-/data-center network functions. (2) Supports advanced middleware for the secure high performance communication between heterogeneous applications. (3) Enables the integration of robotics systems with the FIWARE platform.

1.1.2 The Technical Chapter 1.6: Applications/Services and Data Delivery

As stated in the section 1.1.1 the Applications/Services and Data delivery Chapter (hereunder Apps chapter) is included in the FICORE project and aims at building a framework able to support the creation of an ecosystem of applications, services and data. In this regard, the main objective of the Apps chapter is allowing the monetization, selling and consumption of different kind of digital assets using flexible pricing and revenue sharing models. On the other hand, the Apps chapter also offers a set of tools intended for the creation of highly configurable dashboards and cockpits that can be used for the visualization and management of services and data.

The main functionality provided by the Apps chapter can be grouped in the following major architectural blocks:

- **Business Framework:** This architectural block is responsible for the commercialization of FI products (Apps, services and data). This includes:
  - The publicizing, purchasing and monetizing of products.

  - The search and discovery from different providers of different digital assets, including apps, datasets, APIs, services, etc., that best fit the customer requirements.
  - The management of different business aspects related with the selling of digital assets, including the management of pricing and revenue models, charging and revenue sharing.
  - The accounting of the usage of those products to enable pay-per-use pricing models.

- **Application Mashup Framework:** This architectural block is responsible for the creation of composite FI applications based on highly configurable graphical dashboards and operation cockpits targeted to different devices(e.g laptops, tablets). To support such creations, this block leverages on product search, discovery and comparison as provided by the Business Framework. These capabilities allows FI applications developers to easily find, eventually from different providers, the different required components, including widgets, mashups, services/APIs and datasets.
- **Data Visualization Framework:**This architectural block is responsible for providing dedicated technologies for the dynamic visualization of dataset according to many different paradigms and forms and including high-level customizable analysis capabilities. Data is coming from different sources, both static and real-time, using different formats. This framework provides the building blocks in the form of widgets to the above mentioned frameworks and serves as interface to the GEs developed in the Data/Media Context Management Chapter.

1.1.3 Previous state of the Business Framework

As stated in the section 1.1.1 the FICORE project is the continuation of the FIWARE project. In this way, there was an initial architecture and some initial implementations regarding the FIWARE Business Framework described in section 1.1.2. The current section describes the initial state of the Generic Enablers that build up the FIWARE Business Framework and therefore the starting point of this Master's Thesis.

Following, the components of the Business Framework as well as the functionality they provided at the beginning of the FICORE project are described.

Store GE

The Store GE offers support for selling digital assets to both consumers and developers of Future Internet applications and services, and for end-to-end managing of offerings and sales. In this regard, the Store GE supports the publication of new

offerings, manages offering payment, and provides access to all purchased services or software downloads if the offering includes downloadable assets.

To allow providers to include multiple digital assets in a single offering, the Store GE manages two different concepts:

- **Offerings**: These components represent the abstract concept of an offering, and includes non technical information such as the pricing model or the legal conditions.

- **Resources**: These components are the real digital assets being offered which are included in offerings, taking into account that a resource can be included in multiple offerings and that an offering can include multiple resources. The Store GE supported two different types of resources. On the one hand, the Store GE supported *API resources*, that were created by providing an URL pointing to the endpoint of the given resource. On the other hand, the Store also managed *Downloadable resources* which were directly uploaded to the Store (e.g an application).

For describing the needed information regarding an offering, the Store GE relies on the Linked USDL [19]. The Linked USDL is a RDF-Based vocabulary that allows to describe service offerings, not only at a technical level, but also at a business level (e.g pricing models, legal terms, etc). The Store GE used the Linked USDL just for the serialization and representation of the pricing models that were applied to the given offerings. Therefore, not all the information regarding an offering, such as the included resources, was described in the Liked USDL description.

Finally, the Store GE is intended to support pay-per-use pricing models. To support these models, the Store GE offers an API where the usage information can be gathered in order to calculate the related charges. However, it did not provide accounting functionality in order to monitor the usage of the offered services. Therefore, the Store GE delegated this functionality to the providers of the service.

Repository GE

The Repository GE provides a consistent uniform API to Linked USDL offering descriptions and associated media files for applications of the Business Framework. In this way, the Repository GE is used by the different components of the Business Framework for the storage of the Linked USDL descriptions of the offerings being sold.

The Repository GE allows storing and retrieving Linked USDL descriptions through the provided REST API. These Linked USDL descriptions can be grouped into collections, which can also be nested, allowing to organize them following a

similar approach as in a file system.

It is necessary to take into account that the main functionality offered by the Repository only covered the management of complete documents, that is, the Repository did not parse not interpret the uploaded documents.

Marketplace GE

The Marketplace provides functionality necessary for bringing together offering and demand for making business. These functions include basic services for registering business entities, publishing and retrieving offerings and demands, and search and discover offerings according to specific consumer requirements.

The basic functionality of the Marketplace GE allows to register multiple Store GE instances which can advertise its offerings in order to make them available to a higher number of potential customers. Is important to remark that the Marketplace does not provide features for the acquisition and charging of offerings (provided by the Store GE), but an API that allows to discover existing offerings, as well as, execute full-text searches along the offerings published in multiple Store GE instances. As stated, the Marketplace uses offering descriptions to work. In this way, the Marketplace GE uses the Linked USDL descriptions of offerings, published in the Repository GE, for registering new offerings and for creating the full-text search indexes.

Revenue Settlement and Sharing System GE

The Revenue Settlement and Sharing System (RSS) GE is in charge of distributing the revenues originated by the usage of a given digital among the involved stakeholders. In particular, it focuses on distributing part of the revenue generated by the monetization of a set of assets between the business infrastructure owner and the provider(s) responsible for the service.

The RSS GE calculates how to distribute the revenues generated by the selling of an offering between the provider of the offering and the owner of the Store GE instance where the offering is sold. To execute this calculus the RSS GE requires, on the one hand, a revenue sharing model that specifies the percentage of the revenues that belongs to the Store GE instance owner. On the other hand, the RSS GE requires information about the amount charged to customers of the concrete offerings. The charging information is gathered from the Store GE via API in some documents called *Charging Detailed Records* (hereunder CDRs) which are created every time that a customer is charged. To calculate the revenue sharing, the RSS GE aggregates the charging information contained in the existing CDRs and then

applied the revenue sharing model in order to generate a report with the amount to be paid to the owner of the Store GE instance and the offering provider.

1.2 Motivation of the project

As described in section 1.1.1, the main objective of the FIWARE Platform is helping service providers and entrepreneurs to create context aware applications by providing innovative technological resources and already implemented APIs. In this regard, the FIWARE Business Framework is not intended to be used as an app store, deployed as a global instance, where providers can upload their digital assets to be sold, but to be integrated in the solutions developed by FIWARE users. Concretely, the FIWARE Business Framework aims at providing FIWARE users a set of components that give them a number of search, discovery, monetization and revenue sharing features that could be easily integrated with their own solutions in order to create value added applications.

Taking into account the FIWARE Business Framework state described in the section 1.1.3, it is possible to realize that the features offered by the framework are not enough to cover the business requirements of the users of the FIWARE Platform, since there is a lack on how the different models are created and some features are not mature enough to be used in a real environment.

Evaluating the features and functionality provided by the initial version of the FIWARE Business Framework, it has been identified some points that need to evolve in order to meet the requirements of the users of the FIWARE platform and achieve the objectives of the framework.

First of all, the generated linked USDL documents do not contain all the needed information regarding the offerings and the digital assets being offered. This documents only contain the pricing model and the legal terms. Moreover, the Linked USDL description of an offering represents all the offered resources as a single service, even if the offering contains multiple digital assets. Therefore, there is a mismatch between the information included in the Linked USDL document and the real offering being offered. This blocks the development of search features over offering descriptions since the contained information is not correctly align with the real offerings.

Additionally, the Store GE manages two types of resources, API and Downloadable. This model allows to represent most of the existing digital assets, since most of the digital assets can be described as an URL or as a file. However, this model is too general and some specific information required by some digital assets (e.g meta info contained in an specific type of asset) cannot be provided or managed. In addition, this model impedes to create programmatic validation of the resources, since it is not possible to know what the resource exactly is. It is important to remark,

that this problem might force FIWARE users that want to include monetization features within their solutions using the FIWARE Business Framework, to develop ad-hoc components for validating resources and permissions (e.g check whether the providers have permission to register a backend service) or even modify the sources of the Business Framework components in some scenarios.

Moreover, the Store GE supports pay-per-use pricing models; however, it delegates the accounting to the service providers. This forces FIWARE users, that need to manage pay-per-use services within their solutions, to develop their own accounting component. Moreover, the Store GE needs to trust that the provided accounting information is real.

In addition, the Repository GE allows to store and retrieve Linked USDL documents, but the API only allows to retrieve a complete document whose URL is already known. Moreover, It does not provide any smart search mechanism such as SPARQL to perform queries over the stored Linked USDL documents, so the possibilities and advantages of using a RDF-based serialization mechanism are not being exploited.

Finally, the RSS GE only supports revenue sharing models that involve a fixed percentage between the owner of the FIWARE Business Framework infrastructure and one provider of the monetized assets. This approach is not enough to deal with the complexity of monetizing composite services and applications where multiple stakeholders may be involved, since this kind of models cannot be created. Thus, FIWARE users that want to develop a solution involving multiple partners might have trouble monetizing digital assets.

Taking into account the problems described above, the project developed during the Master's Thesis is required in order to take care of those problems, and provide a FIWARE Business Framework able to support FIWARE users requirements and give them the possibility of enhancing their own solutions with a comprehensive set of monetization, revenue sharing, search and discovery features.

1.3 Objectives of the project

The main objective of the current Master's Thesis project is the enhancing of the FIWARE Business Framework in order to deal with the identified problems, and offer FIWARE users a real solution which could be used to create value added applications by integrating FIWARE Business Framework features within their own projects. The new FIWARE Business Framework features are intended to help digital assets providers to easily monetize their products, giving them support across the whole offering life cycle: from creation, to acquisition, monetization and revenue sharing. Additionally, the new proposed FIWARE Business Framework also focuses in helping potential customers, giving them features that allow to find those offer-

ings that best fit their requirements.

To achieve the main objective of this master's thesis project, some minor objectives have to be accomplished. Concretely, it has been defined the following objectives for the FIWARE Business Framework:

- The FIWARE Business Framework must support the monetization of multiple digital assets including applications, services, APIS, datasets, etc. In this regard, the FIWARE Business Framework must be flexible enough to allow the monetization of almost any kind of digital asset required by the providers, but without creating a too general model that might not allow to control the different resources being offered.

- The different offerings offered thought the FIWARE Business Framework must be described in a RDF-based description language (*Linked USDL* [19]) allowing the semantic annotation of business aspects. The Linked USDL model for describing the offered digital assets must be able to describe not only technical aspects but business level aspects such as the pricing model and the legal terms.

- The FIWARE Business Framework must support composite offerings of assets provided by different providers. In this way, the framework must support the definition and execution of revenue sharing models able to describe how the revenues originated by a composite offering must be distributed thought the involved stakeholders.

- The FIWARE Business Framework must support pay-per-use pricing models. To deal with this kind of models, it is required to make the accounting of the usage of the offered products.

- The FIWARE Business Framework must help potential customers looking for offerings to find the one that best fits their needs. In this regard, the FIWARE Business Framework must provide the means for performing smart searches over offerings of different providers, as well as their comparisons based on predefined criteria.

Finally, the last objective of the Master's thesis project is demonstrating how the FIWARE Business Framework can be integrated in a real environment for the monetization of a concrete software. In this case, by integrating the framework with CKAN (see annex B.3), which is one of the most important open source platforms for the management of Open Data, creating a Data Market. Additionally, this solution also must support the monetization of real time data streams offered by Orion, the reference implementation of the Context Broker GE (see annex B.2).

1.4 Structure of the Rest of the Document

The current document is divided into a series of chapters dealing with the different objectives identified in section 1.3.

The second chapter *State of the Art*, contains an overview of the existing commercial solutions in the area of the electronic commerce. Moreover, it includes a review of the technologies that have been useful for the development of the current project.

The next chapter, *Business Framework Architecture*, describes the general architecture that have been created in order to achieve the objectives of the project. It also includes an overview of the architectures of the different components of the FIWARE Business Framework.

Then, the chapter 4 *Offering Model* deals with the objective of having a RDF-based model for the serialization of offerings describing the model which has been created for this purpose.

The chapter 5 *Multiple Digital Assets Support: The Plug-in Model* describes the solution that have been created to achieve the objective of supporting the monetization of any kind of digital asset within the FIWARE Business Framework.

Additionally, the next chapter, *Pay-Per-Use: Accounting Proxy*, contains the description of the work carried out in order to achieve the objective of supporting pay-per-use models, providing also support for making accounting of the offered assets.

Next chapter, *Revenue Sharing*, describes the solution that have been found to deal with the objective of supporting composite offerings which require to share generated revenues along multiple service providers and stakeholders.

In addition, the chapter 8 *Smart Searches: Marketplace and Repository* covers the work carried out in order to deal with objective of supporting smart searches of offerings.

The chapter 9 *Use Case: The Data Market* describes how the developed FIWARE Business Framework can be applied to enhance external applications, in this case a data management system called CKAN and a service able to give real time data streamings called Orion (implementation of the Context Broker GE), with the features provided by the FIWARE Business Framework.

Finally, chapter 10 *Results and Conclusions* justifies how the developed project satisfies the existing objectives and requirements, and establishes a set of conclusions regarding the work carried out. In addition, this chapter contains a set of future lines that can be applied in a future evolution of the FIWARE Business Framework.

Chapter 2

State of the Art

2.1 Commercial Solutions

In the current section it is evaluated the State of the Art regarding commercial solutions that provide E-Store functionalities. Concretely, existing commercial solutions has been divided into three main categories: *Consumer Centric App Stores*, *Enterprise Stores*, and *Cloud Services Stores*. Each of these sections contains an overview of the main existing products as well as an evaluation of different aspects of these products, such as their business models, possible classifications, etc.

2.1.1 Consumer Centric Apps Stores

Having a look at worldwide app downloads volumes and generated revenues, the app store opportunity remains large. This can be seen by looking at the existing statistics, where there is an estimated revenue of the worldwide app stores in March 2015 of $8,3 billions and a estimated number of downloads by 2017 of 268,692M [2]. Additionally, millions of apps are available in the most important app stores, having 1,4M of apps in Apple App Store [3], 1,5M of apps in Google Play [17], and 340,000 in Windows Store [34] by March 2015.

At least 56 consumer centric app stores were operating in late 2011. The table 2.2 shows a subset of the most prominent ones that have been operating until Q1 2015, and others that passed away and whose occurrence in the table intends to capture unsuccessful business models (e.g Most of them were operated by telecommunication operators).

From the perspective of their ownership, consumer centric app stores can be classified into mobile app stores from device manufacturers, OS and software platform developers (including Java and Chrome), telecommunication operators, component manufacturers, and independent app stores. Mobile application stores are very attractive for these organizations for several reasons. On the one hand, they promise

	Device Manufacturer	**OS Developer**	**Network Operator**	**Component Manufacturer**	**Independent**
Apple App Store [3]	X	X			
Blackberry World [4]	X	X			
Opera Mobile Store [23]	X	X			
Palm App Catalogue	X	X			
Samsung App Store [27]	X	X			
LG Smart World [18]	X				
Dell Mobile Applications	X				
Google play [17]		X			
Windows Phone Market [34]		X			
Verizon VCasst (passed away on January 2013) [31]			X		
T-Mobile Web2Go (Precursor)			X		
Vodafone 360 (passed away on January 2012)			X		
ATT App center			X		
Sprint Digital Lounge [30]			X		
Orange App Shop [24]			X		
Intel AppUp				X	
Qualcom BREW Apps Store			X	X	
MediaTek App Store				X	
GetJar [15]					X
Amazon AWS [1]					X
Mohango					X
Cherokee Market					X

Table 2.2: Consumer Centric App Stores

tremendous growth. On the other hand, mobile application stores are crucial in the competition of mobile platforms.

From the perspective of their business model, they can be classified in those following a closed model, which includes both failed approaches (e.g. Compuserve and KPN iMode) and successful ones (e.g. the Apple App Store, a relatively closed approach), and those following an open model (e.g. Google Play, a relatively open approach). Apparently, the main question for successfully commercializing mobile services is therefore not whether to choose a purely open or closed approach. Other factors like impact of network effects, economies of scale, platform differentiation, quality assurance, and transaction costs have influence on the design of successful mobile app markets (see for example the comparison of inter-organizational business models of mobile app stores performed in [39]).

A different approach, also based on the target business model, is to classify them in three different groups depending on the role played by the service providers. Just as the recent history of mobile content sales, the first app stores were largely driven by two business models commonly known as on-portal and off-portal, both of which require service provider support. With the advent of the so-called *non-portal* App Store-like models, the service provider does not have a front stage role in the content or application sales transactions and is not even required for payment services.

The on-portal business model requires the service provider to manage all aspects of content provisioning, marketing, consumer discovery, content delivery and consumer care. It also requires the service provider to maintain systems and resources to support each of these functions. The common off-portal business model requires the service provider to support a shortcode messaging service (SMS) and a payments service to businesses that market directly to consumers via media such as the web, TV, radio or magazines.

The primary value added to the off-portal business by the service provider is the ability to collect payments from a consumer without extra data entry through the device, a significant benefit when working with mobile devices and impulse sales. To address off-portal business models, service providers either put in place the necessary systems and work directly with third parties or, more often, service providers work through aggregators (like OpenMarket in the U.S.) who carry most of the operational burden, and offer to third parties the extra benefit of consolidating access to multiple service providers within a given market.

With the advent of Apple App Store-like models, a new business model takes shape: the *non-portal* model, where the service provider does not have a front stage role in the content or application sales transactions. Although they benefit from an increase in mobile data usage, they are witnessing the control over more and more elements of the mobile purchasing experience being wrested from their hands

in favour of the on-device content and application stores. The service provider is not even required for payment services. Credit card accounts are provisioned independently with the customers (e.g. using iTunes).

In Apple App Store-like models, developers can easily on board with a simple, straightforward revenue share contract and can rely on the availability of broadband over WiFi or the cellular network. They can access a number of APIs (e.g. iPhone API) including an in-app purchasing API. Most existing application stores have also demonstrated the power of opening the OS directly to developers. Camera, GPS and touch-screen capabilities enhance applications to the extent that customers get "locked into" using the device. In simple terms, App Store-like models unlocked the revenue potential of the mobile application ecosystem, a move that has profited almost every link in the mobile value chain.

The success of a mobile platform depends on the successful design of a viable mobile ecosystem of related services and components. Apple App Store and Google Play remain the two biggest app stores. Those app stores have experienced a tremendous growth during last year (2014), growing at least a 50% [**?**].

There is also a shift from one-off fees to in-apps revenue models. For example, in the case of the Apple App it reached the 76% of the total revenues, while in some Asian countries this value was above the 90% by March 2013, as stated in an article published in Forbes [37]. However, there were still some publishers who were very successful with a one-off fees strategy.

The table 2.4 offers a a comparison of various app store models. The table demonstrates the differentiating factors in today's application store competitive arena. Service providers can bring several valuable assets to create a killer application platform that would be attractive to the developer and content partner community.

Other classification criteria include pricing support (free, single payment, subscription, pay per use), payment options (credit card, paypal, bank account, own proprietary system e.g. Google wallet), supported products (apps, digital services, non-digital products e.g. goods, non-digital services e.g. website design, source code or API access). The table 2.6 offers a comparison of various app stores according to this criteria.

Table 2.6 also shows the recent proliferation of cloud-vendor stores: AWS Marketplace is an online store that allow customers to purchase AMI images with any kind of software ready to be deployed using the EC2 service. VMware solution exchange: is an online store that allows buying VMware virtualization software as well

	Operating System(OS) Specific or On-Device Application Store (e.g Android, Apple, RIM)	**White-Label Application Store**	**Service Provider Application Store**
Third party access to APIs	OS-specific	OS-specific	Service provider-specific
Distribution Channels	Web, On-device Store, Mobile portals, Enterprise program	Web, Partner on-device, Partner Mobile portals, Direct sales	Web, Mobile Portals
Consumer Billing Options	Third Party (e.g PayPal)	Third Party	Service Provider billing
Developer Business Models Supported	Direct application sales, Advertising, In-app content sales	Direct application sales	Direct application sales
Developer Support	Online resources, Conferences and events	Business development, Marketing, Sales, IT	Online resources, Conferences and events
Personalization Support	In some cases	No	No

Table 2.4: App Stores Models Comparison

	Apple App Store	**MS Azure Store**	**Google Play**	**AWS Marketplace**	**VM Solution Exchange**
Free	X	X	X	X	X
Single Payment	X	X	X		X
Subscription	X	X	X		
Pay per use		X		X	
Credit Card	X	X	X	X	
Own proprietary System			X		
Apps	X	X	X		X
Digital Services		X		X	
Non-digital products			X		
Source Code	X				
API Access	X		X	X	

Table 2.6: App Stores Offering Comparison

as applications compatible with VMware products. Microsoft azure store is part of the Microsoft cloud platform and allows purchasing both software to be deployed in the cloud infrastructure and apps for the management dashboard.

2.1.2 Enterprise Store

Corporate mobile technology for enterprise IT is increasingly complex. More and more employees are using personally owned smartphones and tablets at work, and firms are leveraging an array of mobile applications to interact with employees, customers, partners, and suppliers.

Apps downloaded from public App Store-like platforms, both for mobile devices and PCs, disrupt IT security, application and procurement strategies. This impede their alignment with the goal of consumerization of IT, which is to give users what they need while allowing IT to manage corporate policies and procedures.

To manage and control the fragmented mobile application landscape, proactive companies are deploying enterprise app store environments. Companies assemble these environments from three categories of app stores: consumer-focused app stores, vendor-sponsored app stores, and internal corporate app stores.

Enterprise app stores promise greater control over the apps used by employees, greater control over software expenditures and greater negotiating leverage with app vendors, but this greater control will only be possible if the enterprise app store is widely adopted.

The increasing number of enterprise mobile devices and enterprise adoption of mobile device management (MDM) will drive demand and adoption of enterprise app stores. According to Gartner, by 2017 25 percent of enterprises will have an enterprise app store for managing corporate-sanctioned apps on PCs and mobile devices [40].

The Enterprise app store is the cornerstone of mobile device management (MDM) services and represents a different approach to mobile management on its own. Being more end-user centric than IT centric, enterprise app stores can support a more diverse and competitive automated software process requiring less procurement intervention. They offer a way to enable procurement to broaden user choice and to automate the procurement of enterprise software licenses from app stores under corporate control as part of the normal requisitioning process. Indeed, they allow monitoring demand for popular apps that may benefit from better negotiation of license terms and prices.

The long-term success of an enterprise app store hinges on a dramatic increase

in the supply of software solutions. Over the next four years, vendor strategists will expand corporate app store functionality beyond mobile application deployment to also offer comprehensive file sharing, storage, content sharing, and enhanced reporting and monitoring services for smartphones, tablets, and eventually PCs.

2.1.3 Cloud Services Store

Gartner is anticipating a growth in cloud services this. The company estimates that from past 2013 through 2016 $677 billion will be spent on cloud services worldwide [38], which represents an opportunity for emerging stores focused on selling applications and services based on cloud services.

Moreover, all signs point to app marketplaces becoming the standard for small and medium-sized businesses (SMBs) cloud adoption. Ninety percent of the ISVs attending the 2013 Small Business Web Summit were distributing their products through marketplaces, and ten percent of the crowd said that about a third or more of their sales are coming through application stores. Another testament to the momentum of the marketplace model is the high-profile brands that are investing in it. The governments are also eager to reduce costs and to create momentum for ICT SMBs willing to compete with large IT companies for public-sector contracts.

In cloud business, marketing and sales have their own laws. Entirely new sales structures have to be developed and staffed, and some companies have already started to build their own ecosystem and partnerships where SMBs or governmental boties will shop in the future: the "Cloud Store", a marketplace for business customers and a powerful sales channel that will represent the go-to-market for software in the near future: via every channel and with all available resources.

What has become standard for private users now also works in a business environment: users can immediately evaluate, recommend and select a solution that works, without bothering with patches, updates and other time-consuming administrative tasks. On the other hand, software providers can become cloud ready within just a few days, enabling them to expand their market reach without high entry costs or investment. This represents a huge advantage, especially for SMBs, since cloud stores could offer the highest security standards, thus enabling smaller companies to offer higher standards than ever before to their customers. A change in IT procurement has begun with the availability of online catalogue of cloud-based services. Cloud stores will surely drive much faster adoption of cloud services, and provide deeper insights for CIOs and SMEs, creating a more competitive market in the cloud domain.

What follows are three relevant examples of working cloud stores targeted at the public sector and the SMBs respectively:

- CloudStore [5]: Targeted at the UK public sector, CloudStore is one of the

first app store of its kind in the world and the first in the UK for public sector ICT procurement. Cloud App Store (CloudStore) is an initiative of UK government launched in February 2012 in association with G-Cloud that contains all the services currently on the G-Cloud framework (http://gcloud.civilservice.gov.uk/). With the commencement of the third procurement iteration (G-Cloud iii), the Cloud Store was offering more than 800 suppliers and more than 7,000 services across all types of cloud service models, including public, private and hybrid, and under four lots: IaaS, PaaS, SaaS, and Specialist Cloud Services (SCS) such as configuration, management and monitoring. At the time of writing this report, CloudStore has evolved to an online GCHQ-vetted catalogue hosted on Microsoft Azure with more than 8000 SaaS services (distributed amongst 24 categories), 143 PaaS services, 900 IaaS services, and 3497 SCS available for buying the UK public sector. Through this governmental G-Cloud application procurement site, SMEs across the UK get a chance to sell IT services to the public sector alongside major companies, at the same time that delivers savings to the Government. The store offers search services, whose classification is based on domains, ability to create an account to enable governmental bodies to save and export search results, easier browsing through service lots and their sub-categories, a set of filters based on the key features of each services, etc. The Store runs its own open procurement for each of the G-Cloud frameworks. Companies interested in offering their cloud services through the Cloud App Store bid by submitting details and pricing of the services they wish to sell to the Government Procurement Service, which is responsible for awarding places on the G Cloud III Framework. They enter then a process of assurance and accreditation based on CESG (the information assurance arm of GCHQ) to evaluate the Technologies for data security. It can take around three months for a submission to get onto CloudStore, though it depends on the nature of the service and technical issues. Every application or service provides with details like provider contact, license terms, services provided, pricing model, other services from the same provider, etc. What is missing on the store is rating of the provider and its services.

- Fujitsu Cloud Store [14]. The number of solutions from diverse industries targeted at SMBs leads to higher attention to new offerings. Customer reviews and assessments are visible for all offers. Visitors can find solutions quickly and reliably by using intelligent search functions, such as tags. Online marketing and social media activities generate additional visitors while traffic in the store results in additional lead generation for software vendors. Fujitsu is also opening a reseller channel for software partners, thus establishing comprehensive and personal contact between providers and customers. All Fujitsu Cloud Store processes can be used for your own business. The Fujitsu Cloud Store can be used with your own sales engine. Integrating the subscription processes into your website is also planned. In October 2012, just after its launch, the Fujitsu Cloud Store had more than 50 partners, 60 apps (e.g. the

Open-Xchange email and collaboration suite), 250 customers and 700 users.

- White Sky's "My Cloud Store" [33]: is the one-stop-shop for a range of New Zealand's small business software (business, IT and productivity apps). Business solutions are provided as online cloud (SaaS) based services, on a monthly subscription basis. The current offering includes customer relationship Management (salesforce), point of sale (vend), mail security (SMX), payroll (SmartPayroll), project Management (Workflow Max), Mobile device Management (airwatch), online accounting, invoicing, Billings and banking (Xero), data backup (iBus), inventory Management (Unleashed), Office suite (MS Office 365), performance Management (CBS) and job Tracking, scheduling and dispatching (vWorkApp). The store website also offers a directory of regional IT advisors that can assist with mycloudstore applications.

2.2 Service Description Languages

In this section it is described the State of the Art regarding the main existing models for the description of digital services. The different description languages described can be grouped in two different sections. On the one hand, the *technical description languages*, that provide the means for representing the information of a service at a technical level including information about data types, existing interactions, operations performed, etc. On the other hand, the *non-technical description languages* or *Business-oriented description languages*, which allow to describe business level aspects, such as business and pricing models, service level agreements, or legal terms and conditions.

2.2.1 Technical Description Languages

As stated before, a technical description language allows to describe services at a technical level. In this regard, this section contains an overview of two of the most used technical description languages, the *Web Services Description Language* (WSDL) [36] and the *Web Application Description Language* (WADL) [32].

The WSDL is used for describing web services (*SOAP* Services [29]) addressing the need of describing its communications in an structured way. For this purpose, WSDL defines a XML grammar able to describe network services as collections of communication endpoints capable of exchanging messages. Moreover, WSDL service descriptions serve as documentation of distributed systems and can be used as a recipe for automating the details involved in the communications between applications.

A WSDL document defines web services as collections of network endpoints (ports). Additionally, a WSDL document defines the abstract concept of endpoint

and message separated from their actual network deployment or data format binding, allowing to reuse those abstract definitions. In this way, a message is defined as the abstract description of the data being exchanged, and a port type is defined as an abstract collection of operations. The particular protocol and data format for a concrete port type is what is called a binding. Therefore, a port is defined by the association of a network address with a binding, and a collection of ports define a service.

An example of a WSDL document defining a Request/Response communication over HTTP follows:

```
1  <?xml version="1.0"?>
2  <definitions name="StockQuote"
3
4  targetNamespace="http://example.com/stockquote.wsdl"
5            xmlns:tns="http://example.com/stockquote.wsdl"
6            xmlns:xsd1="http://example.com/stockquote.xsd"
7            xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
8            xmlns="http://schemas.xmlsoap.org/wsdl/">
9
10     <types>
11        <schema targetNamespace="http://example.com/stockquote.xsd"
12               xmlns="http://www.w3.org/2000/10/XMLSchema">
13            <element name="TradePriceRequest">
14               <complexType>
15                   <all>
16                       <element name="tickerSymbol" type="string"/>
17                   </all>
18               </complexType>
19            </element>
20            <element name="TradePrice">
21               <complexType>
22                   <all>
23                       <element name="price" type="float"/>
24                   </all>
25               </complexType>
26            </element>
27        </schema>
28     </types>
29
30     <message name="GetLastTradePriceInput">
31         <part name="body" element="xsd1:TradePriceRequest"/>
32     </message>
33
34     <message name="GetLastTradePriceOutput">
35         <part name="body" element="xsd1:TradePrice"/>
36     </message>
37
38     <portType name="StockQuotePortType">
39         <operation name="GetLastTradePrice">
40            <input message="tns:GetLastTradePriceInput"/>
41            <output message="tns:GetLastTradePriceOutput"/>
```

```
42          </operation>
43      </portType>
44
45      <binding name="StockQuoteSoapBinding"
46          type="tns:StockQuotePortType">
47          <soap:binding style="document"
48              transport="http://schemas.xmlsoap.org/soap/http"/>
49          <operation name="GetLastTradePrice">
50             <soap:operation
51                 soapAction="http://example.com/GetLastTradePrice"/>
52             <input>
53                 <soap:body use="literal"/>
54             </input>
55             <output>
56                 <soap:body use="literal"/>
57             </output>
58          </operation>
59      </binding>
60
61      <service name="StockQuoteService">
62          <documentation>My first service</documentation>
63          <port name="StockQuotePort" binding="tns:StockQuoteBinding">
64             <soap:address location="http://example.com/stockquote"/>
65          </port>
66      </service>
67
68  </definitions>
```

A WADL document is used for describing HTTP-based web applications (typically REST services) modelling the resources provided by a service and the relationships between them, by defining a XML-based format. In a WADL document, a service is described using a set of *resource* elements, which contain a set of *param* elements describing the inputs and a set of *method* elements describing the *request* and the *response* of the resource.

An example of a WADL document describing a basic search service follows:

```
1  <?xml version="1.0"?>
2   <application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3    xsi:schemaLocation="http://wadl.dev.java.net/2009/02␣wadl.xsd"
4    xmlns:tns="urn:yahoo:yn"
5    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
6    xmlns:yn="urn:yahoo:yn"
7    xmlns:ya="urn:yahoo:api"
8    xmlns="http://wadl.dev.java.net/2009/02">
9     <grammars>
10      <include
11        href="NewsSearchResponse.xsd"/>
12      <include
13        href="Error.xsd"/>
14    </grammars>
15
16    <resources
```

```
17        base="http://api.search.yahoo.com/NewsSearchService/V1/">
18      <resource path="newsSearch">
19        <method name="GET" id="search">
20          <request>
21            <param name="appid" type="xsd:string"
22              style="query" required="true"/>
23            <param name="query" type="xsd:string"
24              style="query" required="true"/>
25            <param name="type" style="query" default="all">
26              <option value="all"/>
27              <option value="any"/>
28              <option value="phrase"/>
29            </param>
30            <param name="results" style="query"
31                type="xsd:int" default="10"/>
32            <param name="start" style="query"
33                type="xsd:int" default="1"/>
34            <param name="sort" style="query" default="rank">
35              <option value="rank"/>
36              <option value="date"/>
37            </param>
38            <param name="language" style="query" type="xsd:string"/>
39          </request>
40          <response status="200">
41            <representation mediaType="application/xml"
42              element="yn:ResultSet"/>
43          </response>
44          <response status="400">
45            <representation mediaType="application/xml"
46              element="ya:Error"/>
47          </response>
48        </method>
49      </resource>
50    </resources>
51
52  </application>
```

2.2.2 Business-level Description Languages: Linked USDL

While a technical description language focuses on describing services at a technical level, a Business-level Description language focuses on non technical aspects such as the pricing model or the service level agreement. In this regard, this section covers one of the most flexible languages for this purpose: the Linked USDL [19].

The *Linked USDL* is a version of a service description language called *USDL* (*Unified Service Description Language*) which has been developed with the objective of promoting the use of this description language in a web environment. The USDL manages services as economical or social transactions which have some context information attached to them. This context information describes non-technical information, such as pricing models or terms and conditions to be satisfied by the

involved stakeholders when they consume or pay for the service.

The Linked USDL adapts the concepts of the USDL to the principles of semantic web and linked data by providing a set of ontologies which allow the definition of a service in a RDF [26] document. This vocabulary does not define all the concepts of USDL, but relies on existing ontologies such as FOAF [13], GoodRelations [16], DCTERMS [6], SKOS [28], etc. allowing that a Linked USDL document can be easily extensible and adaptable to the needs of the concrete service.

Linked USDL define three different vocabularies in order to describe different facets of a service. The main vocabularies of Linked USDL are: (1) Linked USDL Core, which is the foundation module of the Linked USDL vocabularies and focuses on service descriptions, service offering descriptions, business entities involved in the service delivery chain, and communication and interaction points. (2) Linked USDL Agreement, which is used for capturing service agreements by making use of *WS-Agreement* [35]. (3) Linked USDL pricing, which extends Linked USDL core with a simple vocabulary that allows to describe the pricing model of services and their business model.

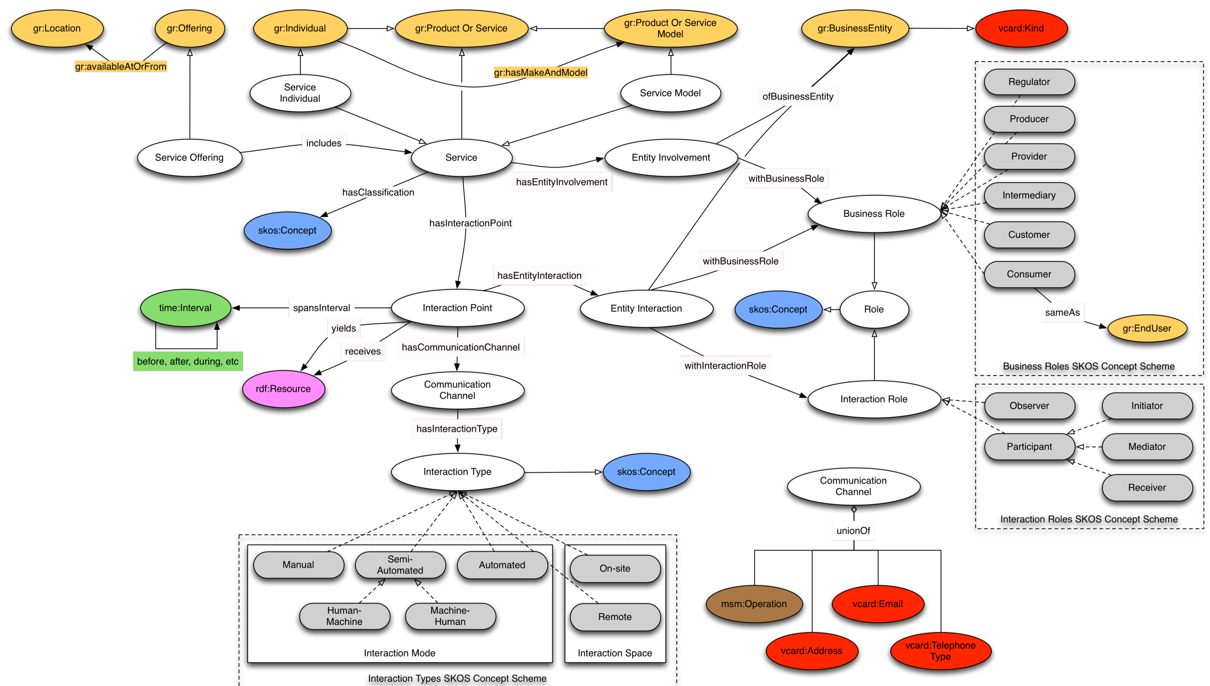


Figure 2.1: Linked USDL Core Vocabulary

2.3 Involved technology

In this section are described the main technologies that have been used for the enhancement of the FIWARE Business Framework. Note that not all the technologies used in the development of the different components are described in this section, but those technologies whose use has significantly increased the value of the FIWARE Business Framework.

2.3.1 RDF

The RDF [26] (*Resource Description Framework*) is a standard specification family of the W3C (*Word Wide Web Consortium*) used as a general method for describing and modelling conceptual information that is implemented as web resources. RDF standard allows the interchange on the Web, providing features that facilitates the data merging even if the underlying schemas differ. The RDF model is based on a series of tuples following a schema subject-predicate-object called triples. This data model is conceptually similar to other models such as an entity-relationship or a class diagram where the subject represents an entity or the object, the predicate represents an attribute or property, and the object represents the value of this property.

RDF is based on the idea of identifying the resources using web identifiers or URIs (*Uniform Resource Identifier*). In this way, a concrete resource is unequivocally identified in the web. It is important to remark that, despite some of these URIs can be URLs which have a real resource associated, not all URIs have to be URLs since it is possible to represent any concept.

In the figure 2.2 it can be seen an example extracted from W3C RDF specification, where its is represented as a graph the information of a person called Eric Miller, identified by the URI http://www.w3.org/People/EM/contact#me. This figure shows how the information is represented using URIs in entities and properties. Moreover, it can be seen how the values of these properties can be *Strings* such as in "Eric Miller" or "Dr".

There are several ways of serializing the RDF, being the first in appearing the XML format, commonly known as RDF since was introduced by the W3C at the same time as the RDF specification. A RDF document serialized in XML is composed by a root node labelled as *rdf:RDF*, where the different namespaces, associated with the vocabularies which are going to be used, are included. This node contains the descriptions of the different resources, labelled as *rdf:Description*, with the attribute *rdf:about* indicating the URI of the resource. Additionally, this description contains the concrete properties and the values of these properties. Following an example of the RDF XML serialization format is shown.

Figure 2.2: RDF Graph representing Eric Miller

```
1  <?xml version="1.0"?>
2  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3       xmlns:contact="http://www.w3.org/2000/10/swap/pim/contact#">
4
5    <contact:Person rdf:about="http://www.w3.org/People/EM/contact#me">
6      <contact:fullName>Eric Miller</contact:fullName>
7      <contact:mailbox rdf:resource="mailto:em@w3.org"/>
8      <contact:personalTitle>Dr.</contact:personalTitle>
9    </contact:Person>
10
11 </rdf:RDF>
```

In addition to the XML serialization format, some text plain serialization formats has been defined in order to make them easier to read. An example of these formats is N-Triples where triples are directly included with their complete URIs, or formats such as Turtle or Notation 3 (N3) which are extensions of N-Triples that allow to include name spaces or to group properties and values. Following it is shown an example of the turtle format.

```
@base <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rel: <http://www.perceive.net/schemas/relationship/> .

<#green-goblin>
    rel:enemyOf <#spiderman> ;
    a foaf:Person ;    # in the context of the Marvel universe
    foaf:name "Green Goblin" .

<#spiderman>
    rel:enemyOf <#green-goblin> ;
    a foaf:Person ;
    foaf:name "Spiderman" .
```

In addition, it has been defined a query language able to retrieve and manipulate data stored in RDF, called SPARQL. This query language allows to create queries against data stored following the schema subject-predicate-object, and provides a full set of analytic query operations such as *join*, *sort*, or *aggregate*.

SPARQL provide different types of queries for different purposes:

- **SELECT**: This queries are used to extract information on a tabular form.
- **CONSTRUCT**: This queries are used to extract information and transform it into a valid RDF document.
- **ASK**: This queries provide a simple true/false result.
- **DESCRIBE**: This queries are used to extract RDF graphs.

Following it can be seen a simple example of a SELECT query that retrieves the names and emails of a set of persons

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?email
WHERE {
  ?person a foaf:Person.
  ?person foaf:name ?name.
  ?person foaf:mbox ?email.
}
```

2.3.2 Open Link Virtuoso

OpenLink Virtuoso [22], has been developed by OpenLink Software with Kingsley Uyi Idehen and Orri Erling as the chief software architects.This system is a SQL-ORDBMS and a Web Application Server that provides SQL, XML and RDF data management in a single multithread server process. Rather that have dedicated servers for each of the aforementioned features, Virtuoso is a "universal server",

which implements multiple protocols. This software allows to access the triple store using different abstractions such as SPARQL, SMILE, Virtuoso/PL, etc. Additionally, Virtuoso is available in open source and commercial editions.

Virtoso is also a OWL Reasoner, which supports a couple of OWL and RDF properties such as *owl:sameAs*, *rdfs:subClassOf*, or *rdfs:subPropertyOf*, etc., used for determining subclasses or property relationships.

In addition, Virtuoso includes a Live SPARQL Query Service Endpoint in all its versions, which can be used for the execution of SPARQL queries over the data contained in the triple store. Moreover, a SPARQL-to-SQL gateway for saving relational data is included in both, the open source and the commercial edition versions.

2.3.3 OAuth2

The OAuth2 [21] protocol aims at allowing third-party applications to obtain limited access to an HTTP Service, both, on behalf a resource owner who has authorized the given application, or by allowing the application to gain access on behalf itself.

OAuth2 introduces an authorization layer and separates the role of the client from the role of the resource owner. In this way, the client requests access to the resources managed by the resource owner and hosted by the resource server. Therefore, the client does not uses the credentials of the resource owner. Instead, the client is given an access token by an authorization server, with the approval of the resource owner, that it uses to access to the protected resources hosted in the resource server.

Figure 2.3 shows the basic OAuth2 flow used to access to a protected resource, including the different interactions between the existing actors:

1. The client requests an authorization from the resource owner. This authorization requests can be made directly as shown or using an external authorization server.

2. The client receives an authorization grant, which represents the resource owner authorization.

3. The client requests an access token by authenticating with the authorization server, and including the authorization grant.

4. The authorization server authenticates the client, validates the authorization grant, and sends an access token.

5. The client requests the protected resource using the access token.

6. The resource server validates the access token and serves the request.

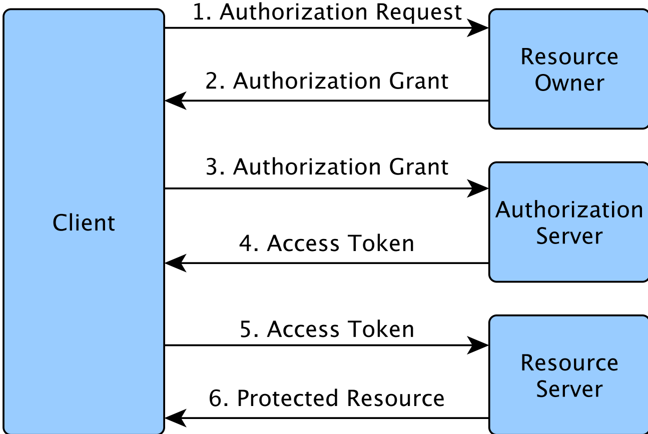


Figure 2.3: OAuth2 Protocol Flow

Chapter 3

Business Framework Architecture

This chapter defines the architecture for the proposed Business Framework, including the relationships between the different components that make up the FIWARE Business Framework and their relationships with third party components that are required for providing the specified functionality. In addition, this chapter also covers the basic architectures of each of the involved systems.

3.1 Architecture Overview

The main objective of the FIWARE Business Framework, as described is section 1.3, is providing FIWARE users a set of search, discovery, monetization, and revenue sharing features that could be easily integrated in their own solutions in order to develop value added applications and systems.

Taking into account the objectives, the FIWARE Business Framework supports the needs of FIWARE users allowing them to create an ecosystem of data, services, and applications by managing the offerings and sales of different kind of digital assets thought the whole offering life cycle: from creation and publication of offerings to monetization and revenue sharing. To provide the required functionality for the FIWARE Business Framework some interrelated components has been identified. Every one of these components focuses on providing a specific part of the functionality:

- The **Store** is in charge of managing offerings, providing monetization and charging capabilities, and dealing with the different pricing models. Is in this component where the digital assets to be sold are registered and where offerings are created.
- The **Marketplace** is in charge of providing search and comparison functionalities. This component allows potential customers to search and compare offerings published in different Store instances, in order to determine the one that best fits their requirements.

- The **Revenue Settlement and Sharing System** (RSS) is in charge of providing revenue sharing capabilities. This component allows to calculate the amount to be paid to the different stakeholders of a given offering or group of offerings, according to the criteria specified in a *Revenue Sharing model* and the charging information provided by the Store.

- The **Repository** is a backend system in charge of managing the RDF description of offerings, written using the linked USDL Vocabulary (See chapter 4 for details on the offerings model). This component provides some APIs for accessing those descriptions and an API for executing SPARQL queries over them.

Despite the different components of the FIWARE Business Framework are able to provide complete functionality by their own, those components have been designed to be integrated and work as a single system, making possible to deploy different instances and providing scalability and flexibility. Figure 3.1 describes the basic interactions existing between the FIWARE Business Framework components.

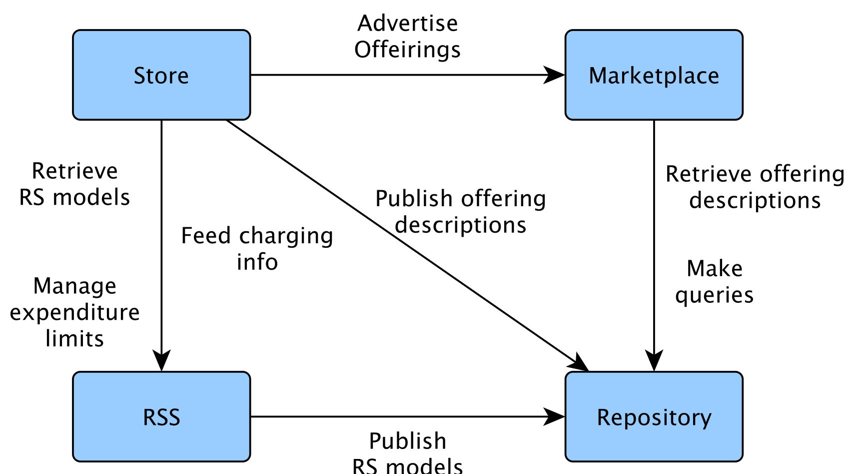


Figure 3.1: Basic relationships between FIWARE Business Framework components

It can be seen in figure 3.1 that FIWARE Business Framework components are highly interrelated, and thus a great deal of integration effort is required. In this way, the Store interacts with the Marketplace, using it to advertise its published offerings. As stated before, offerings are created in the Store, since is the component in charge of offering management; however, offerings providers can advertise its offerings in the Marketplace, making possible that potential customers, who use the Marketplace for the search, discovery, and comparison of offerings, be aware of the existence of their offerings. Additionally, the Store interacts with the Repository by publishing the Linked USDL description of its offerings, making possible that

other components (e.g the Marketplace) could execute queries over them. Finally, the Store also interacts with the Revenue Settlement and Sharing system for three different purposes: (1) Retrieving Revenue Sharing models, in order to allow offering providers to choose the one that will be applied to a given offering. (2) Feed charging information, which is needed by the Revenue Settlement and Sharing System in order to execute the revenue sharing calculus. (3) Use the expenditure limit feature that allows users of the framework to limit their maximum expenses in order to have a better control over them (this functionality is explained in section 3.3.2).

Apart from the Store, the Marketplace and the Revenue Sharing System both interact with the Repository for different purposes. On the one hand, the Marketplace uses the Repository for supporting its search, discovery, and comparison service. In this regard, the Marketplace retrieves from the Repository all the required information of its published offerings, and uses the querying service of the repository in order to allow users to perform smart searches. On the other hand, the Revenue Settlement and Sharing System can optionally use the Repository to publish its Revenue Sharing models in RDF format, making them available as extra information for queries.

It is important to remark, that despite components of the FIWARE Business Framework have been designed to work together, FIWARE users that integrate these components in their solutions are not required to use the complete framework. Therefore, FIWARE users can replace some of FIWARE Business Framework components by their own system or even not including the components if the functionality is not required in the concrete use case.

FIWARE Business Framework components not only interact between them, but also with external components and different type of users. Figure 3.2 describes the FIWARE Business Framework architecture in detail.

As can be seen in figure 3.2, all FIWARE Business Framework components rely on the *Identity Manager GE* (for details on this component have a look at B.1), which provides single sign on and access control features. It is important to remark that the FIWARE Business Framework components are intended to work as a single system. In this regard, it is very important to have a unified system for the management of existing users and organizations as well as their roles and permissions.

Moreover, it is possible to have different instances of the Store as seen in figure 3.2. These instances can be owned by different organizations or providers and are registered in the Marketplace, allowing customers to search and compare offerings that are published and monetized in different places. Additionally, the Store supports pay-per-use pricing models (supported pricing models are described in section 3.3.1). In this way, the Store needs an accounting system, which would generate information on the usage made by the different customers of their acquired assets,

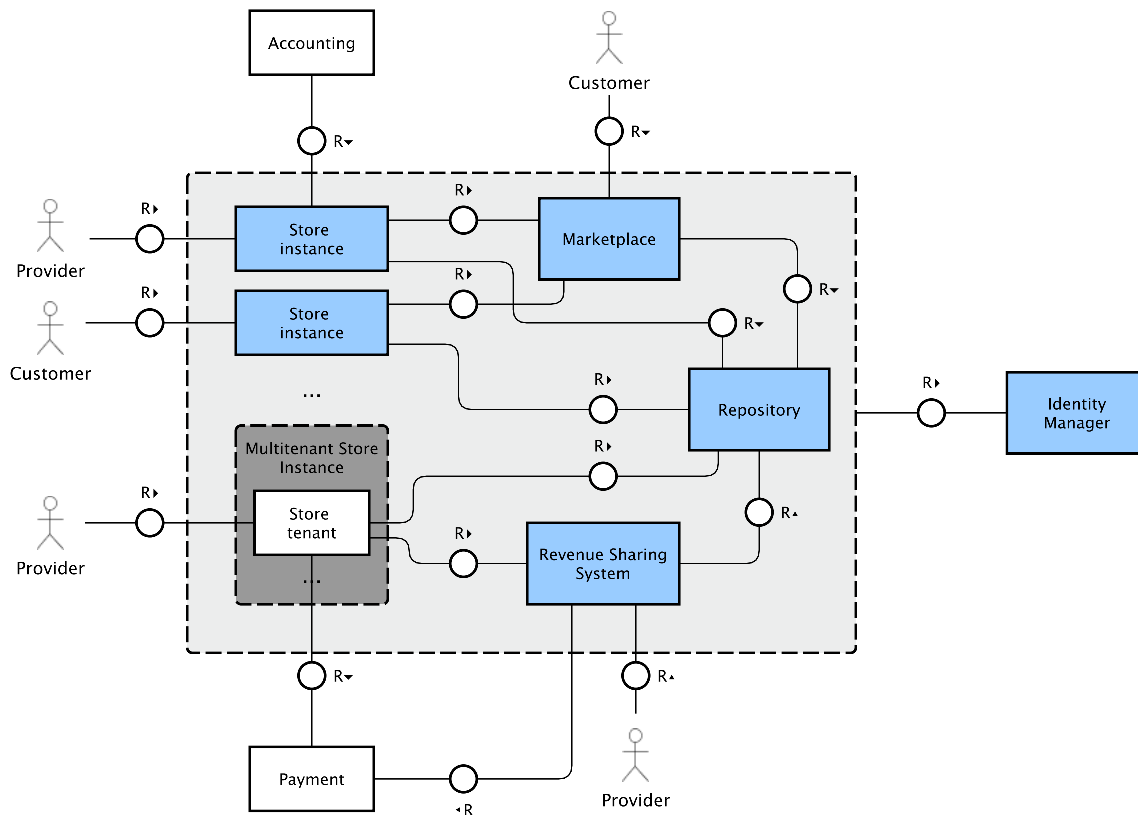


Figure 3.2: FIWARE Business Framework Architecture

in order to be able to charge them (the found solution to deal with accounting and pay-per-use services is included in chapter 6)).

Finally, figure 3.2 also shows the basic roles of the users interacting with the FIWARE Business Framework. It can be seen, that both providers and customers interact with the Store for creating and acquiring offerings respectively. In addition, customers make use of the Marketplace for discovering offerings, and providers use the RSS for the creation of the different Revenue Sharing Models.

The basic architecture of the FIWARE Business Framework has been described, showing the different components that make up the framework and their relationships between them and other third party systems. To end with the overview of the FIWARE Business Framework two basic sample scenarios has been provided, showing the usage of the framework from the perspective of a provider creating an offering and from the perspective of a customer acquiring and using an offering. Note that some important concepts are introduced in these scenarios, for a detailed explanation on those concepts please have a look at the corresponding architecture section.

The first scenario (figure 3.3) describes the end-to-end process associated to a provider creating and offering, and shows the different interactions between a provider and the FIWARE Business Framework components.

The first steps for the provider, once s/he has chosen the digital assets s/he wants to commercialize, take place in the Store. In this regard, the first step consists on registering the different digital assets that are going to be included in her offering. The next step consist on creating the offering skeleton, including its basic info, legal conditions, etc. Then, the service provider is redirected to the pricing editor where s/he can create the pricing model (section 4.2) that will apply to the offering being created. To finish with the creation of the offering, the provider selects the Revenue Sharing Model that specifies how to divide the revenue generated by the offering among the different stakeholders. To do that, the Store retrieves the different Revenue Sharing models previously created by the provider from the RSS. If none of the existing revenue sharing models is adequate, the service provider can access the RSS and create a new Revenue Sharing Model. Note that the Revenue Sharing models are not intended to apply to a single offering but to be used and re-utilized along offerings with similar characteristics.

Finally, once the offering has been created, the provider publishes it in order to make the offering available to potential customers. During this process, the service provider can choose the Marketplaces where the offering is going to be searchable. To allow performing smart searches from the Marketplace, the Store creates a Linked USDL document describing the offering and uploads it to the Repository before registering the offering in the Marketplace.

The second scenario (figure 3.3) describes the end-to-end process associated to a customer buying an offering, and shows the different interactions between a customer and the FIWARE Business Framework components.

The first step in this process is the discovery and comparison of offerings through the Marketplace. Once the customer has decided to acquire a concrete offering, s/he contract it through the store registered on the marketplace where that offering was published. When the Store receives this request, there are two possibilities: (a) In case the offering has at least one *price part* whose price model is different from pay-per-use (e.g. a single payment or the initial payment of a subscription), the Store calculates the corresponding charge. Before charging the customer, it has to check if she has exceeded or not the expenditure limit for a given period of time. This balance of the remaining available expenditure is managed by the RSS, which gives this information to the Store. In case the limit is exceeded with this new charge the process ends and the acquisition is not performed. In the case the limit is not exceeded, the Store charges the customer, then the RSS updates the expenditure balance, and finally the Store creates the required *Charging Detailed Records*

(CDRs) with the related charging information. The Store then forwards these CDRs to the RSS to calculate the revenue sharing. (b) In case the pricing model only includes pay-per-use price parts the process does not calculate an initial charge but the Store performs the necessary steps to allow generation of CDRs later on, based on actual usage of the service.

When the purchased offering includes one or more price parts defining a pay-per-use model, it is expected that an accounting component (i.e. any external component in charge of monitoring the use of the services or any monitoring system operated by the service provider) generates *Service Detailed Record* (SDR) documents with information about service usage made by the customer. Those SDRs will be sent to the charging component of the Store by using a push-oriented accounting API provided by the Store. The Store uses the SDR document and the pricing model of the offering to calculate the amount to be charged associated to usage of service, and generates the corresponding CDRs that are sent to the RSS.

Simultaneously to the generation of the CDR, the Store checks with the RSS that the expenditure limit has not been exceeded. In case the limit is exceeded, the service must be interrupted until the next period of time or treated according the Store policies defined for these cases. Note that the generation of CDRs does not need to take place at the same time SDRs are gathered: generation of CDRs and charging may well be a process that takes places regularly, based on SDRs gathered during a given period of time. The generated CDRs can be later used to generate invoices to customers. Similarly, the RSS makes the Revenue sharing calculation based on the received CDRs and uses the Revenue Sharing model defined for offering to distribute the revenues among the different parties involved.

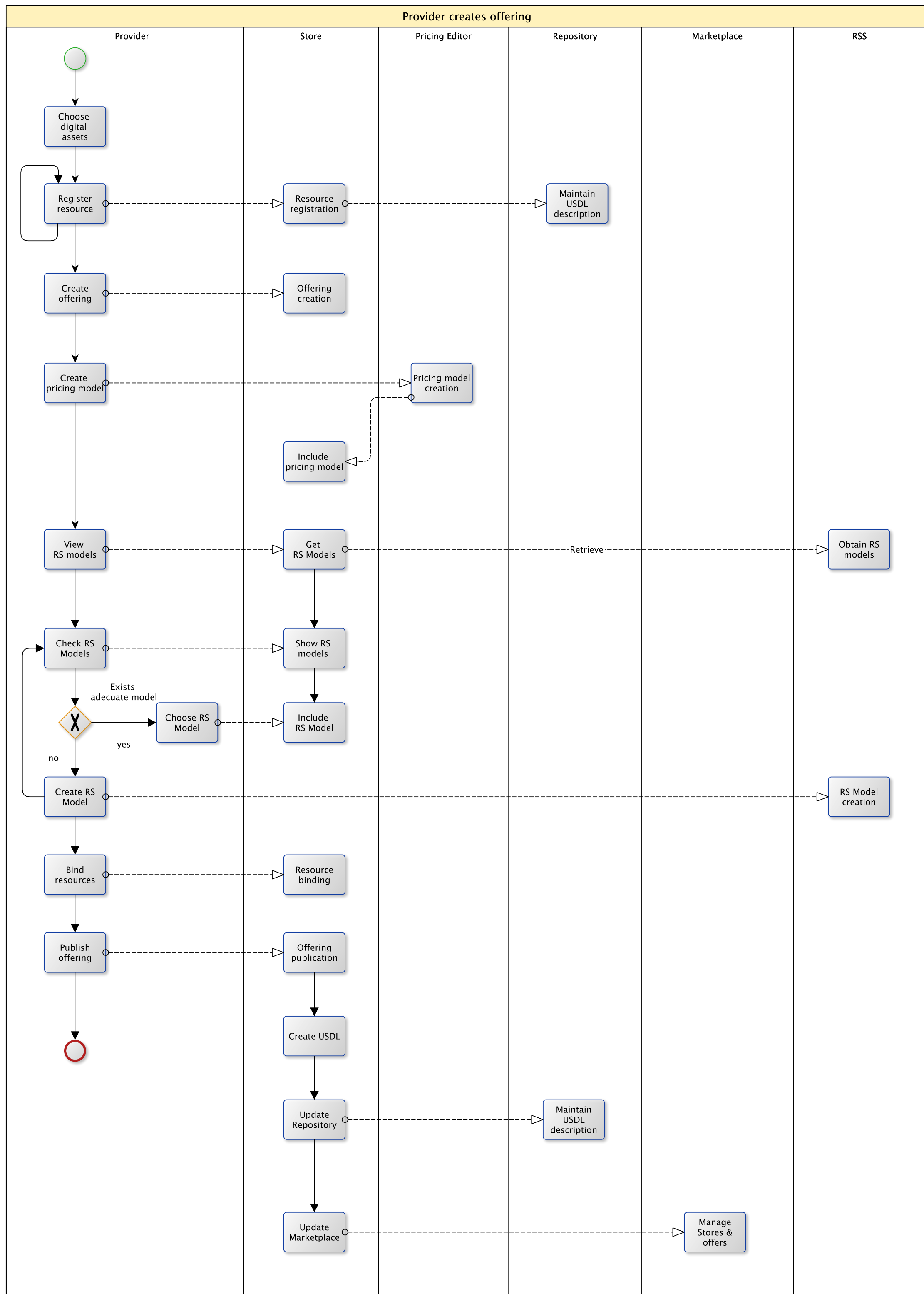


Figure 3.3: Offering creation process

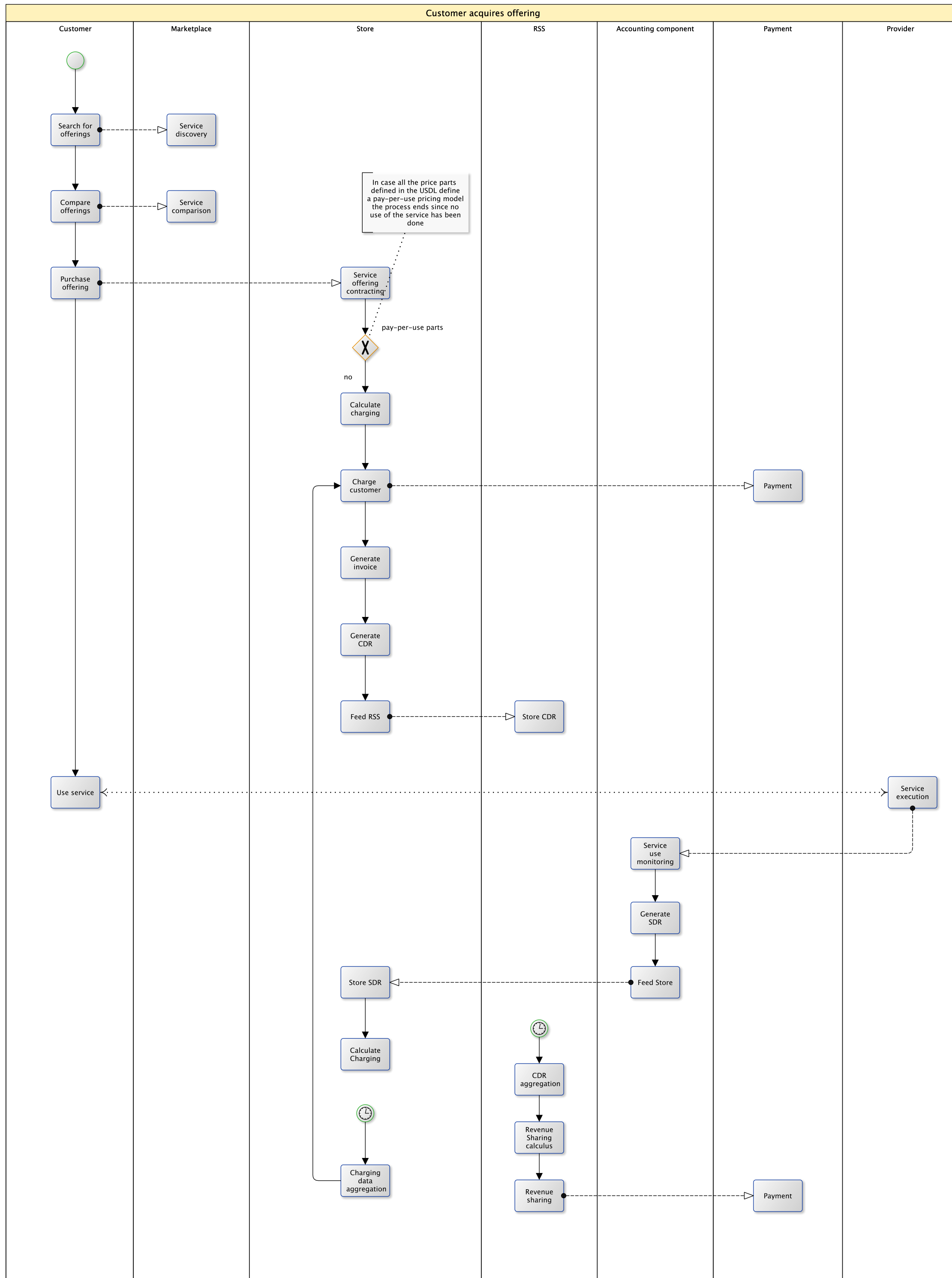


Figure 3.4: Offering acquisition and charging process

3.2 Identity Management and Access Control

As has been stated in section 3.1, the different components of the FIWARE Business Framework have been designed to work together as a single system. In this regard, it is needed to have a unified set of users and organizations within all the related components.

For this purpose, it has been decided to use an external component in charge of managing users, roles, and organizations. Concretely, it has been decided to make use of one of the FIWARE Generic Enablers, the Identity Manager GE and its reference implementation KeyRock (see annex B.1 for details on this software). KeyRock maintains in a single system all the users and organizations who can make use of the FIWARE Business Framework, allowing its components to delegate the authentication of users. This software offers an API where the different components of the FIWARE Business Framework can retrieve the information of an authenticated user including its basic information, the organizations it belongs, or its roles within the component and organizations.

It is important to notice that in the current architecture KeyRock is not used for authorizing users, only for authentication. The authorization of users is made locally in every component of the FIWARE Business Framework using the roles provided by the Identity Manager.

In this architecture, the Identity Manager provides single sign on and single sign out of users using an OAuth2 based approach (see section 2.3.3 for details on this protocol). Therefore, all the components within the FIWARE Business Framework must be registered as OAuth2 applications having their client id and their client secret. However, depending on how the process is launched, the different applications in the framework can be used as the client application or as the accessed service. Additionally, when the different components of the framework are registered in KeyRock, it is also necessary to define the different roles managed by each of the components in order to be able to assign them to users.

Figure 3.5 shows an example of the integration of the components in the FIWARE Business Framework with the Identity Manager KeyRock. Note that in this example, the Store is used as the client application and the Repository as the accessed service; however, this roles may change depending on what component the user access using a web browser, and what component is accessed via API.

It can be seen in the figure 3.5 that the FIWARE Business Framework uses the typical OAuth2 process for authenticating users. Concretely:

1. The user access the web page of the Store.
2. The user is redirected to the login page of KeyRock, including the client id and a callback URL.

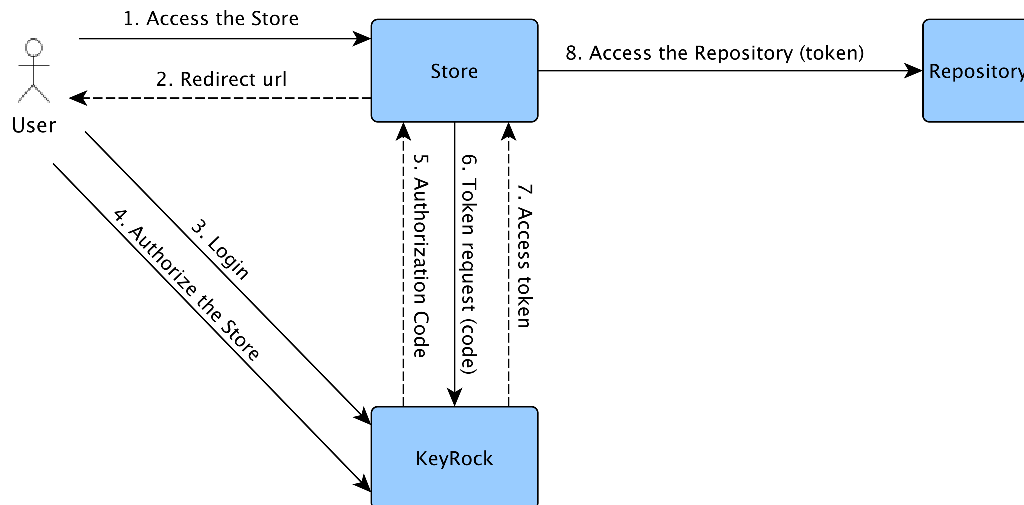


Figure 3.5: Business Framework Authorization Process

3. The user logs in KeyRock

4. The user authorizes the Store to access her data stored in KeyRock

5. KeyRock uses the provided callback URL to redirect the user back to the Store, including an authorization code.

6. The Store requests KeyRock an access token using the authorization code, the client id and the client secret.

7. KeyRock returns the access token that allows the Store to access other component on behalf the user.

8. The Store accesses the Repository using the access token.

3.3 Components Architecture

This section contains the description of the different architectures regarding the components of the FIWARE Business Framework, in order to give an overview of the structure of these components and the main features they provide.

Note that in the different sections some roles, which are applied to the different components, are described. As stated in section 3.2 the components of the FIWARE Business Framework rely on the FIWARE Identity Manager for the management of users, organizations and roles. In this way, none of the described components deals with features such as assigning roles to users or include new users within an

organization. These actions are performed in the Identity Manager which feeds the components with all the required information.

3.3.1 Store Architecture

Within the FIWARE Business Framework, the Store is the component responsible for managing offerings and sales. In this regard, it supports the registration of digital assets, the creation of new offerings, and the management of offering charging and payment. Additionally, the Store is responsible for granting access to services and managing software downloads when new offerings are acquired by customers.

User Model

Taking into account the functionality provided by the Store, a model for managing access and privileges within the component has been defined. This model is intended to be able to determine the possible interactions that users of the Store can carry out. For this reason the following roles has been defined:

- **Admin**: This role is responsible for the administration of the system. Concretely, users with the admin role are able to manage all the information contained in the database, as well as to register instances of the rest of components of the FIWARE Business Framework providing information such as the endpoint where the instance is running.
- **Provider**: This role is responsible for the creation of offerings. That is, users with the provider role are able to register new digital assets and to create new offerings that are then offered to customers.
- **Customer**: This role is intended to allow the acquisition of offerings. In this regard, users with the customer role are authorized to acquire offerings that are published in the Store.
- **Developer**: This role is intended to allow the acquisition of offerings which include a special pricing for developers. Specifically, users with this role are those whose objective is using the acquired offering to develop a new one. Thus, they are authorized to acquire offerings using the special price plans defined for developers.

It is important to remark that roles are not exclusive, which means that users can have multiple roles at the same time (e.g a user can be admin, provider and customer).

The Store supports organizations, which are managed as users groups. In this respect, the Store allows that some offerings might be acquired for all users within an organization, and to publish offerings on behalf those organizations. Additionally, customer, provider and developer roles defined above can also be applied within

an organization. In this regard, a user with the customer or developer role in the context of an organization is able to acquire offerings for the complete organization, and users with the provider role are able to publish offerings on behalf it. Note that users are not limited to a single organization, the same user can belong to any number of them.

Architecture

In a typical app store the different digital assets are directly sold without the possibility of creating value added bundles or monetizing different kind of assets. In this case, the Store deals with this lack by uncoupling the abstract concept of offering from the different digital assets being sold. Concretely, to manage the sales of different digital assets, the Store defines two different concepts: offering and resource.

In this approach, resources are the real digital assets being sold, which can be provided in two different formats. On the one hand, the Store is able to manage digital assets which are accessible using an API (e.g a web service) registering them using their URL. On the other hand, the Store is also able to manage downloadable resources which can be registered by directly uploading them into the system. Additionally, the Store introduces the concept of offering, which is a high level entity that contains business level information, such as the pricing model or the legal conditions. Moreover, an offering includes resources, taking into account that an offering can include any number of resources and that a resource can be included in any number of offerings. This gives providers enough flexibility to create arbitrary complex offerings where multiple digital assets are offered in a single bundle, or to offer the same asset in different offerings with different pricing. As an example, with the described approach providers can create offerings that can include a dataset, the access to a service which is used as a data source, and a visualization tool in a single offering.

At has been already stated, offerings are represented within the FIWARE Business Framework using Linked USDL documents (see chapter 4 for details in the offering RDF model). Taking into account that offerings are created in the Store, Linked USDL documents are also created there, using resource and offering information. It is important to remark that Linked USDL documents format is not user friendly. For this reason, Linked USDL documents are automatically generated using the information given by the provider when registering a resource or creating an offering, making them transparent to final users.

Figure 3.6 shows the Store architecture, which is divided into a series of functional modules. This figure also shows the different relationships of the Store with the rest of components of the FIWARE Business Framework, already covered in section 3.1.

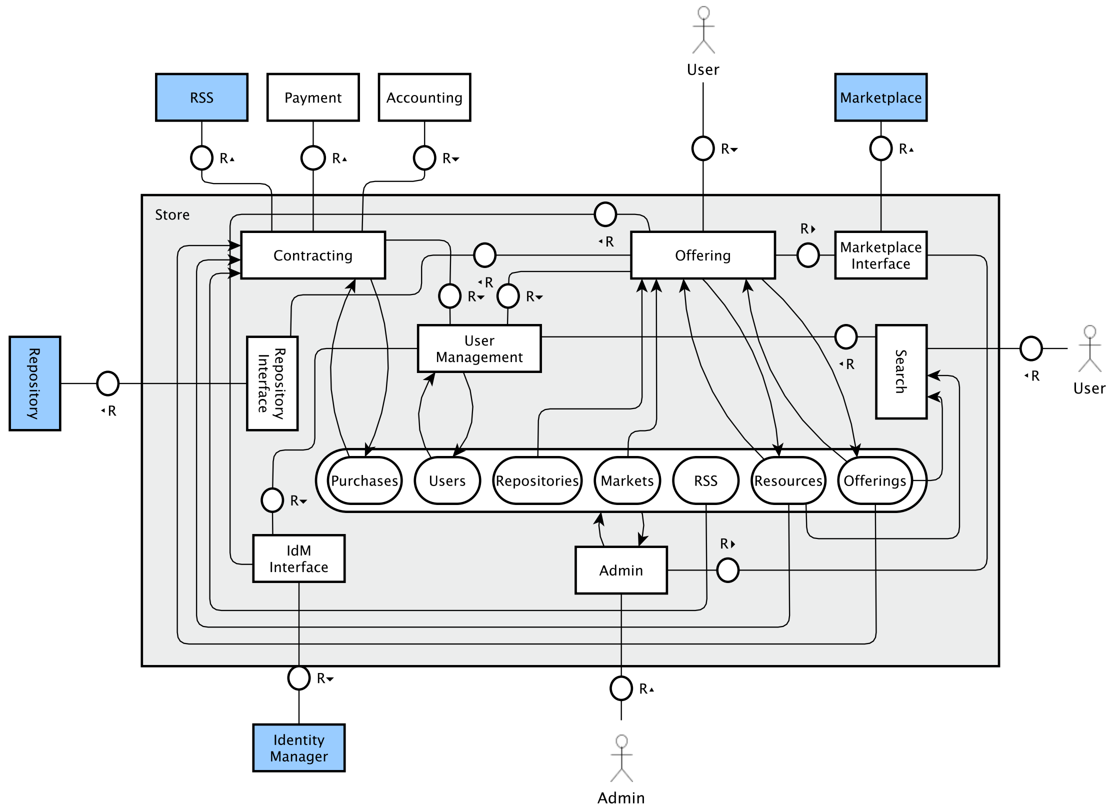


Figure 3.6: Store Architecture

As shown in the figure 3.6, the Store is divided into modules. Each module has a specific functionality and is connected to the other modules and external systems. Following it is described the main functionality provided by them.

The *Marketplace Interface* module is responsible for the communication with the Marketplace in order to register and remove the concrete Store instance, as well as advertise offerings. This module communicates with both the *Offering* module, responsible for the advertising of offerings, and the *Administration* module, which is responsible for registering the Store instance.

The *Repository Interface* module is responsible for communicating with the Repository in order to manage Linked USDL documents associated with the published offerings. This module communicates with the *Offering* module, which is the one in charge of the management of offerings, and thus, the one that requires to upload, update or remove Linked USDL descriptions.

The *IdM interface* module is responsible for the communication with the Identity Manager in order to authenticate users and obtain information about users and

organizations, as well as the different roles of those users.

The *Admin* module is used by users with the Admin role to manage the Store and to register the concrete Store instance in the instances of the Marketplace. This module reads and writes to all data models and contacts the *Marketplace Interface* module for the Store registration.

The *Offering* module is used for the management of offerings. In this respect, this module creates new offerings, registers new resources, binds resources to offerings and puts offerings up for sale. This module is also responsible for the management of acquired offerings and gives customers access to the information on these offerings and their bound resources. This module reads and writes from the *Resource* and *Offering* models and reads from the *Marketplace* and *Repository* models to retrieve the information required to manage descriptions and advertise new offerings. Additionally, this module contacts the *Marketplace Interface* module to send requests to manage the different offerings that are advertised in the Marketplace. It also communicates with the *Repository Interface* module, which it uses to manage the USDL descriptions of the different user offerings. Finally, this module contacts the *User Manager* module, which it uses to check the roles and permissions of the users sending the requests.

The *User Manager* module is responsible for managing users by supervising the different roles and privileges in order to control access to different functionalities of the Store. This module reads and writes to the user model and is contacted by all the modules that are accessible from outside the Store, that is, by the *Offering*, *Search* and *Contracting* modules. This module retrieves user information and roles from the *IdM interface*.

The *Contracting* module is responsible for managing subscriptions and purchases of the different offerings published in the Store. This module contacts different payment gateways, receives payment confirmation, gives access to the digital assets and pass on the charging information (CDRs) to the Revenue Settlement and Sharing System. This module is also responsible for renewing services governed by a subscription payment model. (If service subscription is possible from outside the Store, then the customer have to explicitly notify the Store of subscription renewal via the purchases API.). Moreover, This module is responsible for displaying acquisition information to users. In case the pricing model comprises pay-per-use models, an external accounting component is required to provide service usage information (SDRs) to the Store in order to be able to calculate the amount to be charged. This module contacts the payment gateway, notifies service providers, and sends information on charges to the Revenue Settlement and Sharing System for distribution among stakeholders.

Finally, the *Search* module is responsible for searching published offerings de-

pending on parameters and filters provided by users. This module reads from the *Offering* and *Resource* models in order to get the information required for searches.

The Store is designed to be accessed in several ways. First, the Store provides a Web interface that allows the existing users of the system to carry out all the possible interactions using a web browser. Additionally, the Store offers a REST API which can be used by developers in order to integrate their own solutions with the monetization features offered by the Store. Figures 3.7 and 3.8 describe the existing APIs for the offering, contracting and search modules.

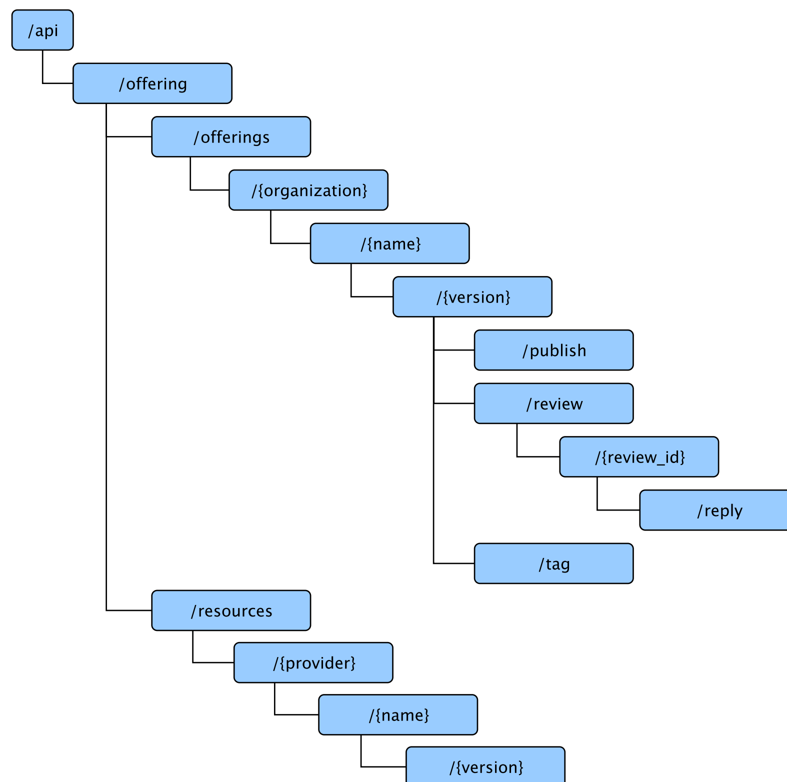


Figure 3.7: Store API Offering module

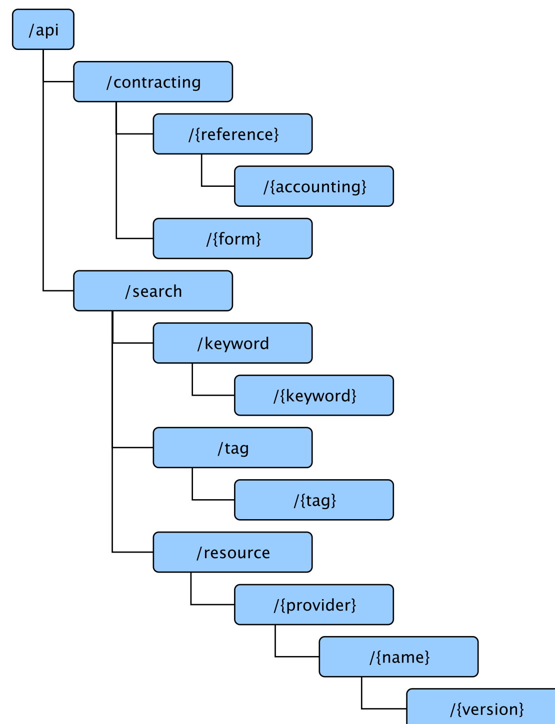


Figure 3.8: Store API Contracting and Search modules

Offering Life Cycle

The figure 3.9 shows all the possible offering states from the viewpoint of a provider. First, the provider creates the offering, whose state is set to *Uploaded*. An *Uploaded* offering is still not up for sale, and is visible only to the user who created it (or other users with the provider role is created on behalf an organization), so the provider can register any number of resources in the Store without having to modify an offering that might have been acquired. Moreover, the information associated with the *Uploaded* offering is editable. The offering moves to the *Bound* state when resources are linked to an offering. The offering is *Bound* for as long as it is not put up for sale and it is linked with resources. If all the resources of the offering are unbound, then the offering returns to the *Uploaded* state. When the provider decides to put the offering up for sale, the later moves to the *Published* state. When the offering is published, the provider can select an existing Marketplace instance and advertise her offering there. Note that a *Published* offering is no longer editable, nor is it possible to modify its information, or bind or unbind resources. If a provider wants to modify information about a *Published* offering, a new offering with a bigger version has to be created. This means that there might be multiple offerings bound to a given digital asset, both in parallel and over time (in this latter case working like versioning). Finally, the provider can delete the offering at any time. However, this action has different effects depending on the state of the offering. If the offering

has not yet been put up for sale, that is, it is *Uploaded* or *Bound*, the offering is simply removed from the Store as it is available to the provider only. If the offering state is *Published*, it moves to the *Deleted* state and is no longer visible to future customers. Note that any customer that has already acquired the offering would still have access to its resources.

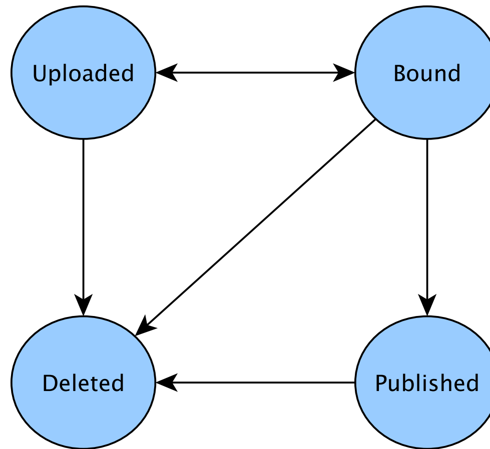


Figure 3.9: Offering Life Cycle Provider Viewpoint

The figure 3.10 shows all the possible offering states and the associated transitions from the point of view of a customer. *Published* is the first state of an offering from the viewpoint of a customer. In this state, customers can view all the information associated with the offering, which they use to decide if acquiring the offering or not. An offering that a customer decides to acquire moves to the *Acquired* state. This state indicates that the customer has purchased the offering, so the purchasing process is complete, and the customer can access or download the resources associated with the offering depending on the type of digital asset that have been included. Finally, an offering that customers no longer require, for example, a subscription to which they no longer want to be subscribed, returns to the *Published* state and can be acquired again by a customer.

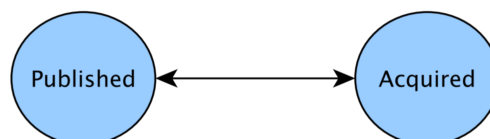


Figure 3.10: Offering Life Cycle Customer Viewpoint

Offering Monetization

This section describes how the Store manages pricing models and monetization at a high level, for details on the offering and pricing models have a look at chapter 4.

As the Store is the component in the FIWARE Business Framework whose objective is managing offerings and sales, it determines how pricing models have to defined in order to monetize digital assets. In this regard, the Store supports the following three basic models, which are used as the building blocks for the creation of complex pricing models:

- **Single payment**: Defines a charge that is made once at the time of acquiring an offering.
- **Subscription**: Defines a periodic charge. In this case the Store charges the customer when acquiring the offering and then periodically depending on the renovation period.
- **Pay-per-use**: Defines a charge that depends on the actual usage made by the customer of the offered services.

With the objective of having complex and flexible pricing models, the Store supports its creation based on putting together a couple of small building blocks, each defining one of the basic models described above, grouped into *price plans*.

A given offering created in the Store can define multiple price plans. In this way, a price plan defines a complete pricing model that can be chosen by customers when they are acquiring the offering. For example, an offering may define a price plan whose model is based on pay-per-use and another price plan based on subscriptions. In this example, a customer who plan to make a big usage of the offering, may choose the subscription based plan which defines a fixed amount to be paid periodically, while a customer who is not going to make an extensive usage of the offering might select the pay-per-use plan.

Additionally, the Store allows providers to include two special price plans which are not acquirable by all the customers. On the one hand, the Store allows to define an special plan intended for updating offerings. In this case, only customers who have acquired a previous version of the given offering can choose this price plan when acquiring the offering. On the other hand, the Store allows to include an special price plan intended for developers. In this case, only users with the developer role can acquire the offering. This allows providers to give better conditions to customer that are going to use the given offering for the development of a new service.

In order to define the pricing model within a price plan, it uses *price components*. A price component is the building block of the pricing models and defines a basic charge based on one of the three basic models supported by the Store (single

payment, subscription, and pay-per-use). To create a new price plan, a provider can include any number of price components. In this model, providers typically will include a price component per resource; although providers are not restricted to this approach.

In addition, price plan can also contain discounts. A discount is an special price component based on pay-per-use that instead of adding an amount to the final price to be charged, it takes away the related amount. Note that discounts have been only defined for pay-per-use models since it does not make sense in single payments or subscriptions. Dining a single payment discount or a subscription discount means that there will exists a fixed amount that is then subtracted from the final amount. In this case this can be achieve just be not including this amount in the definition of the pricing model.

Depending on the price components included in the price plan selected by a customer, the Store calculates charges in a different way. When the user acquires an offering, if it includes single payments or subscriptions, it aggregates the pricing included in those components an charges the customer. For subscription price components it also determines the renovation period, where the customer is charged again. If the pricing model defines pay-per-use components, the Store calculates charges periodically. In this case, if the model defines subscriptions, pay-per-use charges are calculated every time a subscription is renovated. Otherwise, pay-per-use charges are calculated periodically depending on the configuration of the Store.

As stated before the Store is able to support pay-per-use pricing models; However, for this kind of pricing models, the Store requires usage information in order to be able to calculate the amount that has to be charged to the different customers. To deal with this task, the Store offers an API where an accounting component can feed accounting information by providing *Service Detailed Record* (SDR) documents describing user usage information of a concrete service. For details on how the pay-per-use is managed within the FIWARE Business Framework, have a look at chapter 6.

To end with the pricing and monetization overview, the Store supports a notification system that allow service providers to be notified when a customer acquires their offerings. In this regard, when a provider creates an offering she can optionally include a notification URL, which is then used by the Store to send the concrete notifications. This notifications are created as an HTTP POST with the following structure:

- **Offering**: This field is used to identify the offering that have been acquired. For this purpose, this field include the organization, the name and the version of the offering.

- **Reference**: This field contains the reference that have been generated by the

Store to identify the concrete purchase. This value is required for example when feeding accounting information for pay-per-use services.

- **Customer**: This field is used to identify the customer that have acquired the offering. Note that this field can contain the id of a user or of an organization depending of the type of purchase.
- **Resources**: This field contains the list of resources included within the offering that have been acquired. For every resource, it includes the name, the version, the resource type, the media type and the URL of the resource. Note that if the resource has been uploaded to the Store the resource URL points to the endpoint where the resource can be downloaded from the Store.

3.3.2 RSS Architecture

In the context of the FIWARE Business Framework, the RSS is the component in charge of distributing the revenues originated by the monetization of a given digital asset, among the involved stakeholders. Concretely, it focuses on distributing the revenue generated by digital assets between the FIWARE Business Framework providers responsible for the monetization and offering discovery services, and the providers responsible for the asset. It is necessary to take into account that the RSS deals with the complexity of having multiple providers or stakeholders involved in the revenue sharing process.

User Model

Despite other components of the FIWARE Business Framework such as the Store or the Marketplace are accessible by all the registered users, the RSS only can be accessed by admins and offering providers. In this regard, the RSS is intended to be a privileged component within the FIWARE Business Framework, since its main features are intended to be used by providers of the monetized digital assets.

In order to control the access and manage the interactions between users of the FIWARE Business Framework and the RSS the following roles have been defined:

- **Admin**: Users with the admin role are able to manage all the features provided by the RSS and all the information contained in the database without restrictions.
- **Aggregator**: This role is used to represent users that are admins of the systems in charge of generating charging information such as the Store. Aggregators are able to manage the different features provided by RSS limited to the scope of the concrete system they are administrating, for example launching the revenue sharing calculus of all the transactions generated in a given Store instance. Note that users with the admin role in the Store (see section 3.3.1) will have the aggregator role in the RSS.

- **Provider**: This role has a exact matching with the provider role defined in the Store. In this case, users with the provider role are able to manage their Revenue Sharing Models, that are then used to distribute the revenues generated by the offerings they have created in the Store.

Architecture

The RSS distributes the revenues originated by the selling of a couple of digital assets; however, the RSS do not distribute revenues at an offering level. In this respect, the RSS defines an identifier called *Product Class*, which is used to identify an arbitrary set of offerings (owned by a concrete provider in a concrete Store instance) whose generated revenues are expected to be distributed in the same way and between the same stakeholders.

In order to be able to calculate the distribution of revenues, the RSS needs to know who are the involved stakeholders and how to distribute the revenue among them. To deal with this issue, the RSS requires providers to create *Revenue Sharing Models*. These models, are identified by a product class and specify the stakeholders and the algorithm used to calculate the amount to be paid to them. The Revenue Sharing Models structure as well as the way they are managed in the FIWARE Business Framework are described in detail in the chapter 7.

Apart from the Revenue Sharing Models, the RSS requires to know the amount that have been charged to customers regarding the offerings included in a given product class. To retrieve this information, the RSS exposes an API to gather *Charging Detailed Records* (CDRs). These documents are generated by the Store every time it charges a customer and contain the following information:

- **Product Class**: This field contains the product class of the transaction in order to identify the revenue sharing model to be applied.
- **CDR source**: This fields identifies the system that is generating the charging information, that is, the concrete Store instance registered in the RSS.
- **Correlation number**: This field contains a correlation number used to ensure that no charging information is lost between transactions.
- **Time Stamp**: This field describes the time and date when the transaction has been generated.
- **Offering**: This fields is used to identify the Offering which has been charged.
- **Event**: This field specifies the type of event that generated the charge. In this way, this field can contain single payment, subscription, or pay-per-use.
- **Purchase code**: This field contains the reference used to identify the purchase in the Store.

- **Description**: This field contains a textual description of the charged transaction.
- **Cost**: This field contains the amount that have been charged in the transaction.
- **Currency**: This field contains the actual currency used in the transaction.
- **User**: This field identifies the customers who has been charged.
- **Provider**: This field identifies the provider of the offering which have been charged.

Once the RSS have received charging information, the revenue sharing calculus can be launched in several ways. In this regard, it can be launched periodically in order to calculate the revenue sharing distribution of transactions that are pending since a give threshold, or it can be launched manually by admins or aggregators. It is necessary to remark that admins can launch the revenue sharing calculus for the pending transactions of any provider, while aggregators only can launch the revenue sharing calculus for transactions originated in the Store they administrate.

When the revenue sharing calculus has been launched for a set of transactions, the first step carried out by the RSS is aggregating CDR documents by its product class. To do that, the RSS retrieves all the CDRs that use a given product class and calculates the total revenue for them. Once the total revenue for every of the involved product classes has been calculated, the RSS identifies the revenue sharing models to be applied (identified by a product class). Finally, the RSS applies the algorithm described in the different revenue sharing models for each of the product classes in order to calculate the different amounts to be paid to the existing stakeholders.

Figure 3.11 shows the architecture of the RSS as well as its relationships with other components of the FIWARE Business Framework.

It can be seen that the RSS is divided into two main modules. On the one hand, the module in charge of the revenue sharing process, including the management of the CDRs, the Revenue Sharing models and the Settlement process. In this regard, this module is composed of a number of modules which expose a couple of APIs where the Store can feed CDRs and Revenue Sharing models, that are then saved to the database. In addition, this module also contains the *Settlement* module, which is in charge of executing the CDR aggregation and the revenue sharing calculus as described above.

Additionally, the RSS includes a module which is not in charge of any part of the revenue sharing process, but to provide a different functionality to the components

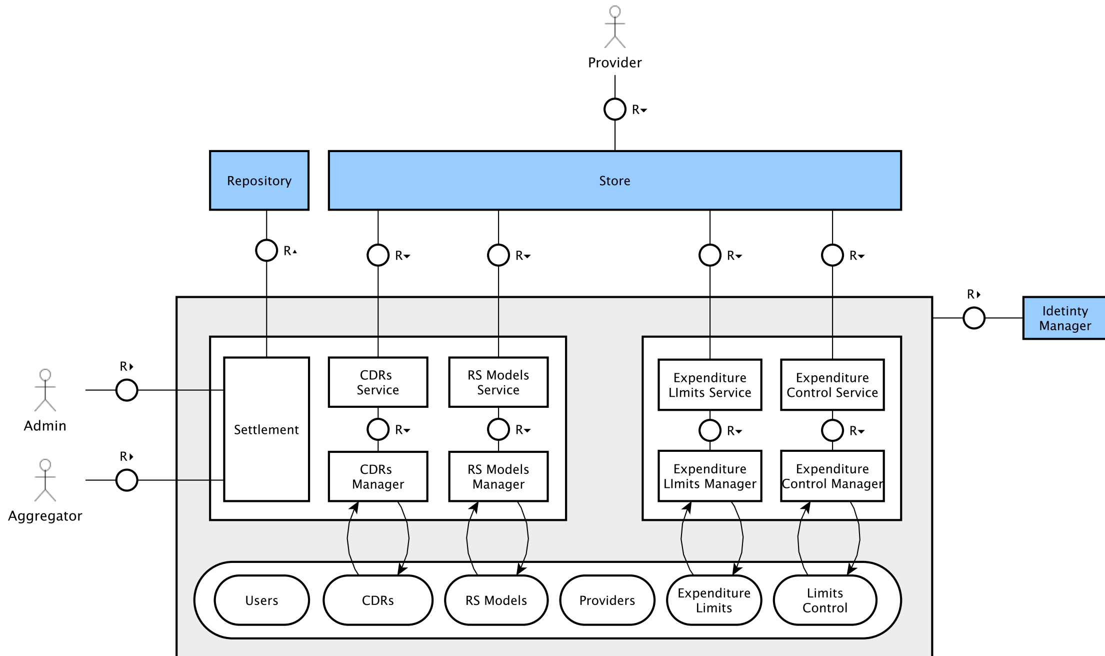


Figure 3.11: RSS Architecture

of the FIWARE Business Framework, the expenditure limit feature. This functionality is intended to allow customers within the FIWARE Business Framework, to define limits on the amount they want to expend in a given period (per transaction, daily, weekly, monthly, etc).

To be able to handle the expenditure limits, the Store is fully integrated with this feature, so if a customer has defined expenditure limits for a given period, the Store uses the API exposed by the RSS in order to check if a user can be charged without exceeding those limits. Note, that if the expenditure limits have been exceeded the concrete customer will not be able to acquire new offerings until the next period. In case the charge is produced by the renovation of a subscription and it exceeds the defined expenditure limit, the customer can choose whether the limit can be exceeded or not. In case the customer chooses that an expenditure limit cannot be exceeded for subscriptions, the subscription is not renovated resulting in losing the access to the related digital assets until the next period.

In addition, expenditure limits can have been exceeded by a charge generated by a per-per-use model. In this case, the Store charges the customer since the usage of the assets has been already made, and revoke the access of the customer until the next renovation period.

In order to interact with the RSS there are two possibilities. On the hand, the RSS provides a web interface that can be used to manage the revenue sharing pro-

cess, allowing to create revenue sharing models, to register Stores (or other external charging components), and to launch the settlement process for a set of transactions depending on the roles of the users. On the other hand, the RSS exposes an API for both for the revenue sharing process and for the expenditure limits management. Figures 3.12 and 3.13 shows the structure of the APIs of the revenue sharing module and the expenditure limits module respectively.

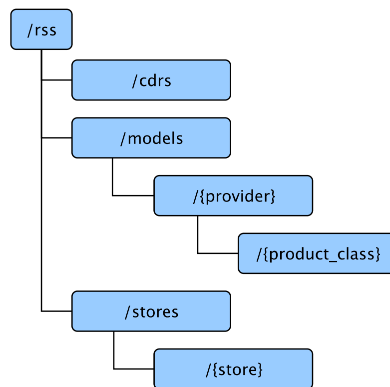


Figure 3.12: RSS API Revenue Sharing module

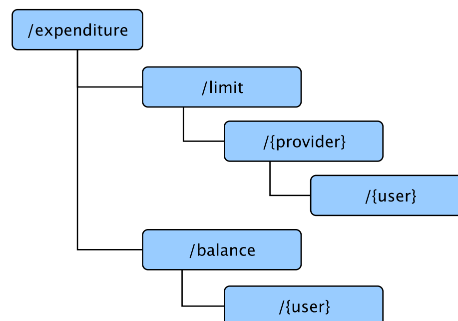


Figure 3.13: RSS API Expenditure Limits module

3.3.3 Marketplace Architecture

Within the FIWARE Business Framework, the Marketplace provides functionality intended to bring together providers and customers. To do that, the Marketplace offers an interface whose objective is allowing customers to discover offerings from different providers and sources (e.g published in multiple Stores), using different mechanisms, from simple and smart searches, to reviews, ratings and comparisons.

User Model

As it happens with other components of the FIWARE Business Framework, the Marketplace defines a set of roles in order to manage the permissions and possible interactions of users with the Marketplace features. The Marketplace defines the following roles:

- **Admin**: Users with the admin role are able to manage all the functionalities provided by the Marketplace as well as all the information contained in the database.
- **Provider**: Users with the provider role are able to advertise its offerings published in a given Store. Note that these users will require having the provider role in the Store, since this role is needed for creating offerings.
- **Customer**: Users with the customer role are able to search and compare offerings that are being advertised in the Marketplace. This role is provided to all the users of the framework by default.
- **Store Owner**: Users with this role are those that have the admin role in the Store, and allows them to register the Store instance they are owning in the Marketplace in order to permit its customers to advertise their offerings.

The Marketplace has support for organizations. In this regard, the Marketplace allows organizations to advertise its published offerings in the same way they create offerings in the Store. To deal with this issue, the Marketplace allows to assign the provider role in the context of a given organization, in order to offer its owners control over what users can advertise offerings on behalf it.

Architecture

Figure 3.14 describes the architecture of the Marketplace. In can be seen that the Marketplace is divided into three main modules in charge of providing different functionality.

First of all, the main objective of the *Registry and Directory* module is managing information on Stores instances. To deal with that, this module allows admins as well as Store owners to register, update, and delete information about Stores in order to support its providers to advertise their offerings.

Additionally, the *Offering and Demand* module gives support to providers to advertise their offerings published in a registered Store instance, in order to allow customers to discover them. Moreover, this module also focuses in giving customers features for searching and comparing the advertised offerings to find the one that best fits their requirements. Therefore, the *Offering and Demand* module brings

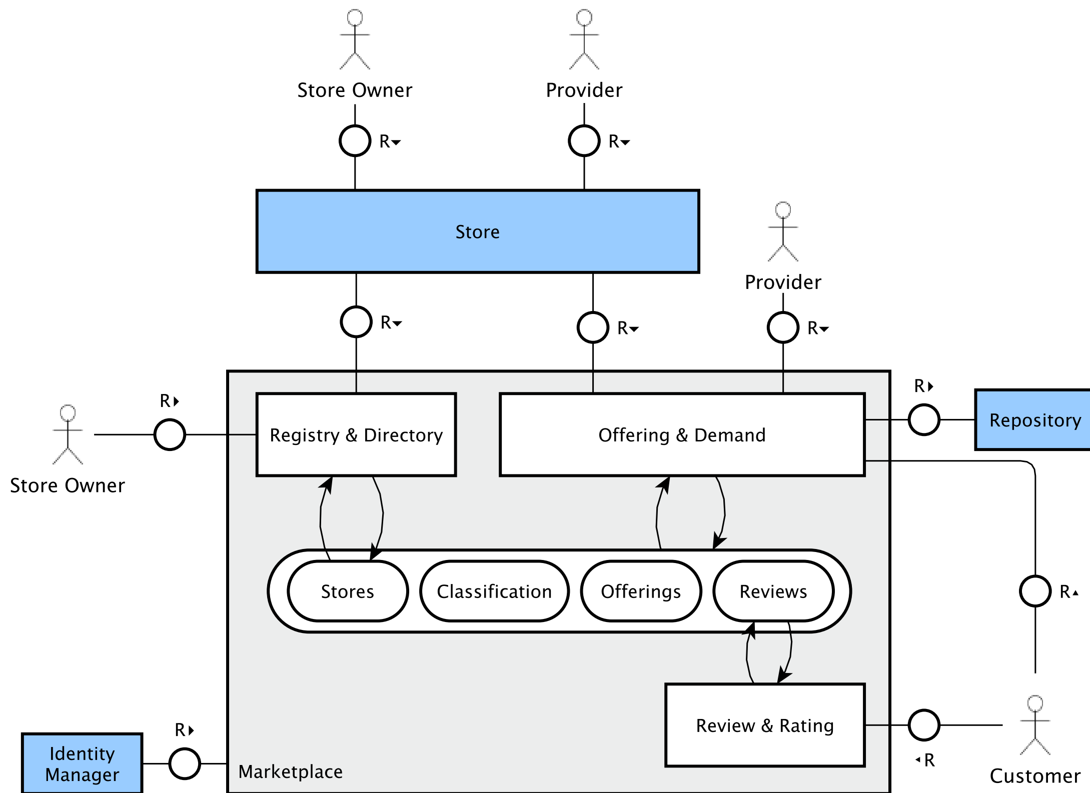


Figure 3.14: Marketplace Architecture

together customers and providers by matching existing offering and demand.

It is possible to see in figure 3.14, that the *Offering and Demand* module makes use of the Repository. Specifically, this module uses the Repository in order to retrieve the different offering RDF descriptions (Linked USDL documents as described in chapter 4) and to execute queries required by the different customers. Note that the Store generates those descriptions when a new offering is published and uploads them to the Repository in order to make information of the existing offerings available.

Finally, the *Review and Rating* component allows customers to give textual and star-rating feedback for offerings and Stores along predefined categories. This rating and feedback can allow other customers to determine the quality of the existing offerings and Stores while they are searching.

Users of the FIWARE Business Framework have several possibilities in order to interact with the Marketplace. On the one hand, it is possible to use the provided Web interface in order to access all the features provided by the Marketplace using a browser. On the other hand, the Marketplace offers an API that can be used to

integrate Marketplace functionalities with other components. Figure 3.15 describes the APIs of the Marketplace.

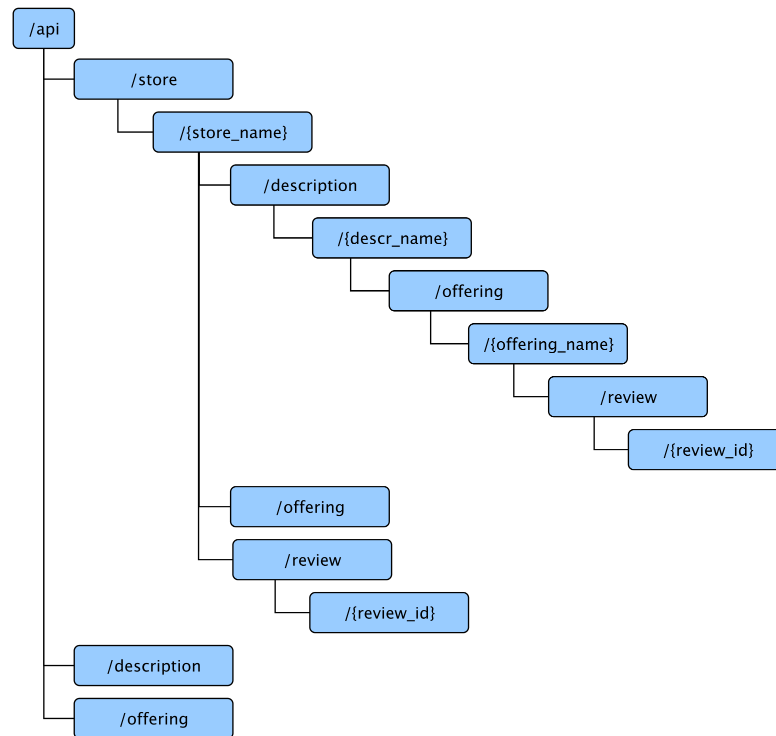


Figure 3.15: Marketplace API

3.3.4 Repository Architecture

Within the FIWARE Business Framework, the Repository is the component in charge of the storage of the RDF descriptions of offerings (Linked USDL documents) and other related media files. To deal with this feature, the Repository provides a consistent API, which other components of the framework (or external ones) can use in order to upload and download descriptions. Moreover, the Repository provides support for making queries over the stored information regarding offerings and digital assets.

User Model

In order to manage the different permissions of the users of the FIWARE Business Framework and the way they can interact with the Repository, it defines some roles that determine the actions that can be carried out. Specifically, the following roles have been defined:

- **Admin**: User with this role are able to perform all possible actions with the Repository and to manage all the information contained in the database.
- **Provider**: This role allows users to upload, update and remove descriptions in the Repository.
- **Customer**: Users with this role are able to download descriptions and execute queries in the Repository. Note that this role is given to all the users of the FIWARE Business Framework by default.

Note that the customer role is given to all the users of the FIWARE Business Framework by default, that means that the published descriptions can be downloaded for all the users. Therefore, the Repository is not intended to maintain private information of the existing offerings, but public information intended to make users understand what is offered and under what conditions.

Architecture

The Repository works with two different objects in order to manage its uploaded descriptions, *Resources* and *Collections*. In this approach, resources are the concrete descriptions which are being managed in the Repository, and that are uploaded, updated, and removed, on behalf the providers of the digital assets offered in the FIWARE Business Framework, from the Store and the RSS. Moreover, resources are downloaded from the Marketplace on behalf customers of the FIWARE Business Framework.

On the other hand, collections are containers for collecting resources, taking into account that multiple collections can be used on the Repository for various purposes. Moreover, Collections can be nested, so providers are offered a similar interface for organizing their descriptions as in a file system. In this regard, collections might be used to keep all content that is locally referred from the descriptions together in one place. For example an offering description often has additional documentation, depictions and other collateral information, which can be bundled together in a single collection.

Figure 3.16 describes the architecture of the Repository. It can be seen, that the Repository allows to negotiate the format in which the information of the resources and collections can be retrieved depending of the needs of the customer. It is important to remark that for negotiating the format of the content of a resource (the description itself) it is needed that this content might be independent of the format and that the Repository might know how to manage the required format. For example, a description uploaded in RDF XML format might be retrieved in turtle of in N3 format.

Finally, the Repository also allows to perform queries over the information included in the existing uploaded descriptions. Note that the format of the result of

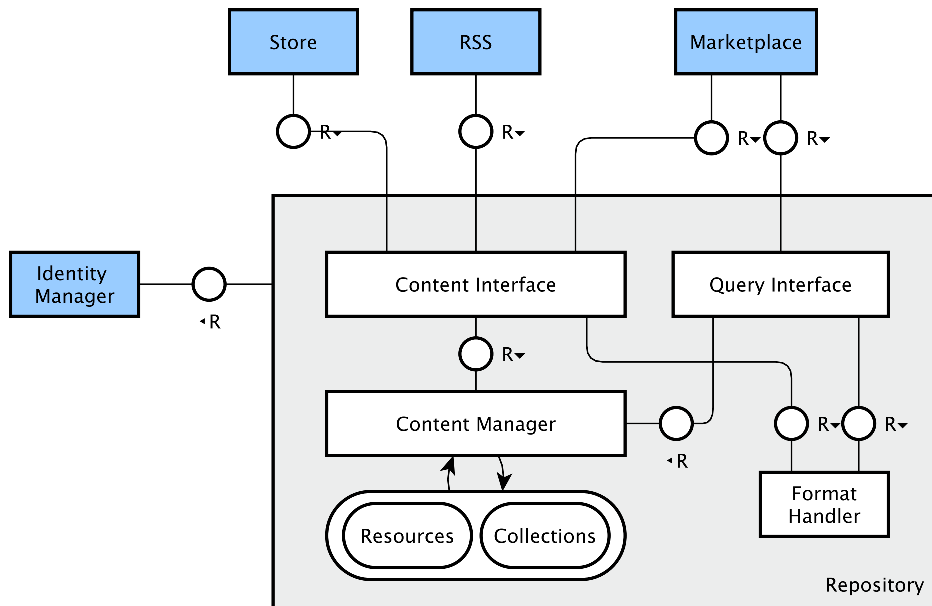


Figure 3.16: Repository Architecture

these queries can also be negotiated. The detailed description on how the querying mechanism works can be found in chapter 8.

Unlike the rest of components of the FIWARE Business Framework, the Repository is a backend system which does not provide a Web interface (Although it is possible to negotiate HTML as type when accessing resource or collection info). Thus, the Repository is intended to be accessed using the APIs it provides described in figure 3.17.

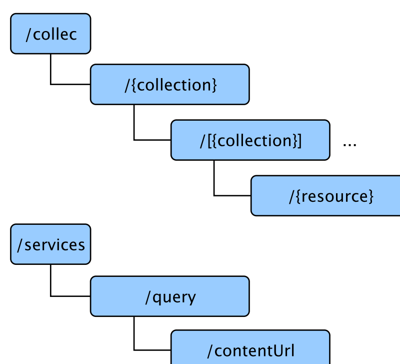


Figure 3.17: Repository API

Chapter 4

Offering Model

As stated in section 1.3, one of the main objectives of the project is using a RDF-based language for describing the different digital assets and offerings, in order to allow the semantic annotation of business aspects and the execution of smart queries. In this way, this chapter describes the offerings model that have been created for this purpose.

This chapter includes some examples of the described models. In order to make them easy to read, they have been written in turtle format.

4.1 Model overview

The offerings model have been created using different RDF vocabularies, but focussing on the Linked USDL family (see section 2.2.2). Concretely, the following vocabularies has been used:

- **Linked USDL Core**: This vocabulary is used to represent basic offering concepts, such as *Service*, *ServiceOffering*, etc.
- **Linked USDL Price**: This vocabulary is used for the creation of the pricing models.
- **DCTERMS** [6]: This vocabulary is used to describe meta data of the different elements. In this case, it is used to represent some general concepts such as names and descriptions.
- **SKOS** [28]: This vocabulary is intended to support knowledge organization schemas such as thesauri or classifications. In this case, it is used for describing types of resources in order to classify the different digital assets.
- **Good Relations** [16]: This vocabulary describes offerings and other aspects of e-commerce in the web. In this case, it is used to describe some business information that cannot be completely described using the linked USDL vocabularies, such as the legal conditions or part of the pricing.

- **PAV** [25]: This vocabulary is used for tracking provenance, authoring and versioning. In this case, it is used to to describe versions of offerings and assets.

- **FOAF** [13]: This vocabulary is intended to link people and information using the Web. In this case, it is used to describe information on providers properties and images.

As stated in section 3.3.1, the selling of digital assets is managed in the FIWARE Business Framework using offerings and resources. In this approach, a resource is the real digital asset being sold (e.g a program, an API, etc.) which is uncoupled from the abstract concept of offering. An offering represents the product, which can include multiple digital assets (resources), and describes business aspects such as legal terms or pricing models. It is important to remark, that the same resource can be included in different offerings. To uncouple the resources from the offerings and allow smart searches over these resources, it is needed different Linked USDL descriptions for the resources and the offerings with a binding between them.

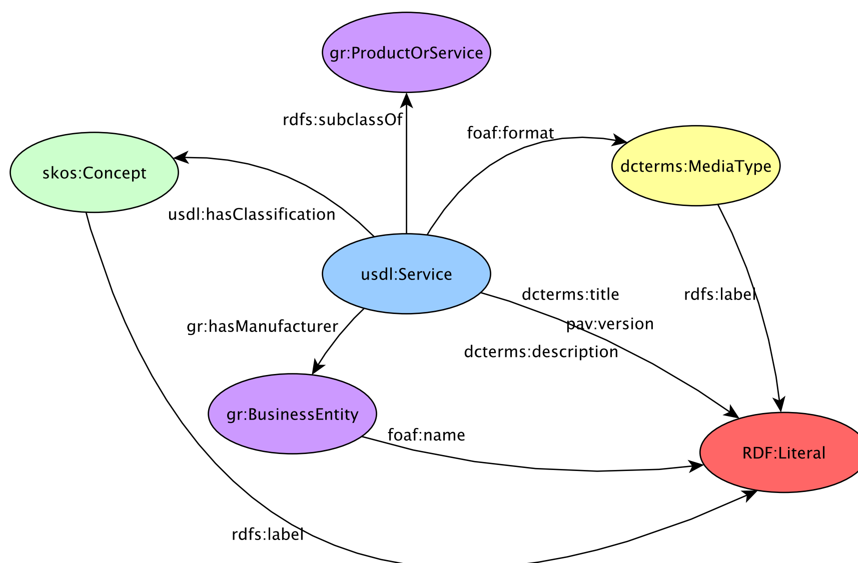


Figure 4.1: Resource USDL model

Figure 4.1 shows the basic model used to represent resources. It can be seen in figure 4.1 that a resource is represented using Linked USDL, concretely, a *usdl:Service* class identified by an URI which is created as an URL pointing to the real resource registered in the Store. The class *usdl:Service* is built as a subclass of Good Relations *gr:ProductOrService*.

The *usdl:Service* has a series of properties describing the information that is required by the FIWARE Business Framework in order to manage resources. Concretely:

- The resource basic info is described using DCTERMS and PAV. In this regard, *dcterms:title* and *dcterms:description* properties are used to include the name and the description of the resource respectively. On the other hand, *pav:version* is used to represent the version of the resource.
- The resource type is represented using *skos:Concept* class which is labelled with the concrete type and linked to the service with the *usdl:hasClassification* property. It is important to remark that this resource type is not intended to describe a media type, but the type of digital asset such as API, Dataset, Widget, etc.
- The media type of the resource (e.g application/zip) is described using the *foaf:format* property pointing to a *dcterms:Mediatype* class, which is labelled with the concrete media type.
- The owner of the resource is described using the *gr:BusinessEntity* with a property *foaf:name* for including the name of the provider. The provider is linked to the resource using the *gr:hasManufacturer* property.

Following, it is possible to find an example of the Linked USDL document of a resource serialized in turtle format.

```
1  <http://store.fiware.org/api/offering/resources/fdelavega/cdata/1.0>
2      a usdl#Service ;
3      dcterms:description
4          "Some testing data being sold" ;
5      dcterms:format
6          [ a dcterms:MediaType ;
7              rdfs:label  "text/csv"
8          ] ;
9     dcterms:title
10         "CData" ;
11    gr:hasManufacturer
12        [ a gr:BusinessEntity ;
13            foaf:name  "fdelavega"
14        ] ;
15     pav:version  "1.0" ;
16     usdl:hasClassification
17         [ a skos:Concept ;
18             rdfs:label  "CKAN Dataset"
19         ] .
```

The model described in figure 4.1 allows to represent the information required by the FIWARE Business Framework about the different digital assets. To be able to monetize those assets, it is also needed to have a model for describing offerings and business aspects. Figure 4.2 contains the basic model used to describe offerings.

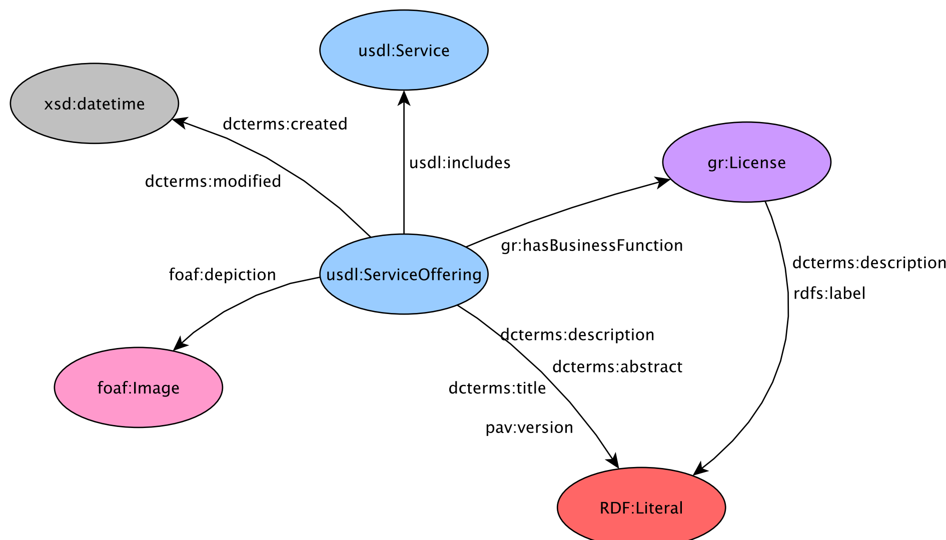


Figure 4.2: Offering USDL model

It can be seen that an offering is described using the linked USDL core vocabulary, specifically the class *usdl:ServiceOffering*. In this case, the URI used to identify the concrete offering is created using the URL of the offering in the Store. This allows potential customers to directly access the concrete Store instance just having the offering description.

It can be seen that the *usdl:ServiceOffering* class has a couple of properties used to describe all the information regarding the concrete offering. Concretely, offering information is described as follows:

- The different digital assets offered in the offering and described as *usdl:Service* instances, are bound using the property *usdl:includes*. In this way, it is possible to describe what are the concrete digital assets being sold in a given offering.
- Some DCTERMS properties are used in order to describe general info of the offering. In particular, *dcterms:title* is used to represent the name of the offering, *dcterms:description* and *dcterms:abstract* are used to represent the textual descriptions of the offering (a long and a short description respectively). Moreover, *dcterms:created* and *dcterms:modified* are used for describing the dates when the offering was created and last modified.
- The version of the offering being described is represented using PAV, specifically the *pav:version* property.
- Offerings contain a main image which is used as a logo. In this case, this image is described as a *foaf:Image* instance and linked to the offering node using *foaf:depiction* property.

- Legal terms and conditions of the offering are also described in this model. In this regard, *gr:License* is used. In this approach *rdfs:label* and *dcterms:description* are used for describing the title and the text of the legal conditions respectively.

Following, it can be seen an example of a basic model describing an offering.

```
1  <http://store.fiware.org/api/offerings/offering/fdelavega/example/1.0>
2    a usdl:ServiceOffering ;
3    dcterms:title "Example"@en ;
4    dcterms:description "This is an example offering"@en ;
5    dcterms:abstract "Example offering"@en ;
6    dcterms:created "2015-06-01"^^xsd:datetime ;
7    dcterms:modified "2015-06-01"^^xsd:datetime ;
8    usdl:includes <http://store.lab.fi-ware.org/ns#3RDPhqrUiClCDh> ;
9    usdl:hasPricePlan <http://store.lab.fi-ware.org/ns#1111111111> ,
10     <http://store.lab.fi-ware.org/ns#HQsRGctt8S1oH3jtB> .
```

Note that figure 4.2 only describes the main information of the offering without containing the pricing model. The basic structure which is used to create pricing models can be found in figure 4.3.

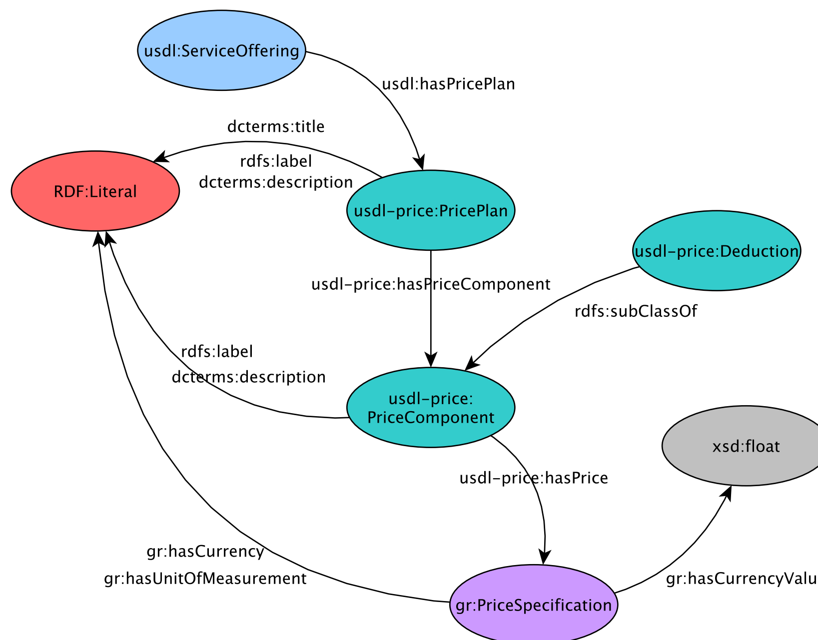


Figure 4.3: Basic Pricing USDL model

It can be seen in figure 4.3, that the main class used in the pricing model is the *usdl-price:PricePlan*, which is linked to *usdl:ServiceOffering* using the property *usdl:hasPricePlan*. It is important to remark that an offering can contain multiple price plans, each describing a pricing model that can be chosen by a customer. The

details on how the pricing models are actually interpreted by the Store in order to charge customers is described in detail in section 4.2.

In this approach the class *usdl-price:PricePlan* has three basic properties describing meta info: (1) *dcterms:title* contains the display name of the price plan. (2) *rdfs:label* is used to represent a label which is used to identify the concrete price plan. (3) *dcterms:description* is used to represent the description of the price plan.

Moreover, the concrete pricing information is described using instances of the *usdl-price:PricePlan* (or *usdl-price:Deduction* which is subclass). Instances of this class are linked to the concrete *usdl-price:PricePlan* instance using the property *usdl-price:hasPriceComponent*. In order to describe the meta info of the *usdl-price:PriceComponent* class, the *rdfs:label* and *dcterms:description* properties are used. These properties contain the label identifying the component and its description respectively.

Finally, the price associated to a price component is described using the class *gr:PriceSpecification* and linked to *usdl-price:PriceComponent* using the property *usdl-price:hasPrice*. To represent the concrete price, this class uses a couple of properties. Concretely, *gr:hasCurrency* is used to represent the currency of the price being described, *gr:hasUnitOfMeasurement* is used to represent the unit of the price (e.g per month, single payment, etc.), and *gr:hasCurrencyValue* contains the concrete value of the price.

Figure 4.3, describes the price component based pricing. However, the model also support more complex pricing definitions intended to support pay-per-use models which depend on multiple variables. In this regard, the model supports price functions that allow to deal with this complexity. Figure 4.4 shows the basic structure of a price component based on price functions.

It can be seen in Figure 4.4, that the main class for representing price functions is *spin:Function* which is linked to the concrete *usdl-price:PriceComponent* instance using the property *usdl-price:hasPriceFuntion*. Additionally, it is possible to provide a textual description of the price function within a price component using the property *usdl-price:hasTextFunction*.

Moreover, the different variables that are used in the price functions, are represented using the classes *usdl-price:Usage* and *usdl-price:Constant*, both subclasses of *usdl-price:PriceVariable*. In this approach, the *usdl-price:Usage* class represents an accounting variable whose value depends on the actual usage made by customers, while *usdl-price:Constant* represent constant values, described using the *gr:QuantitativeValue* class.

The detailed specification on how price functions are created and interpreted can be seen in section 4.2.

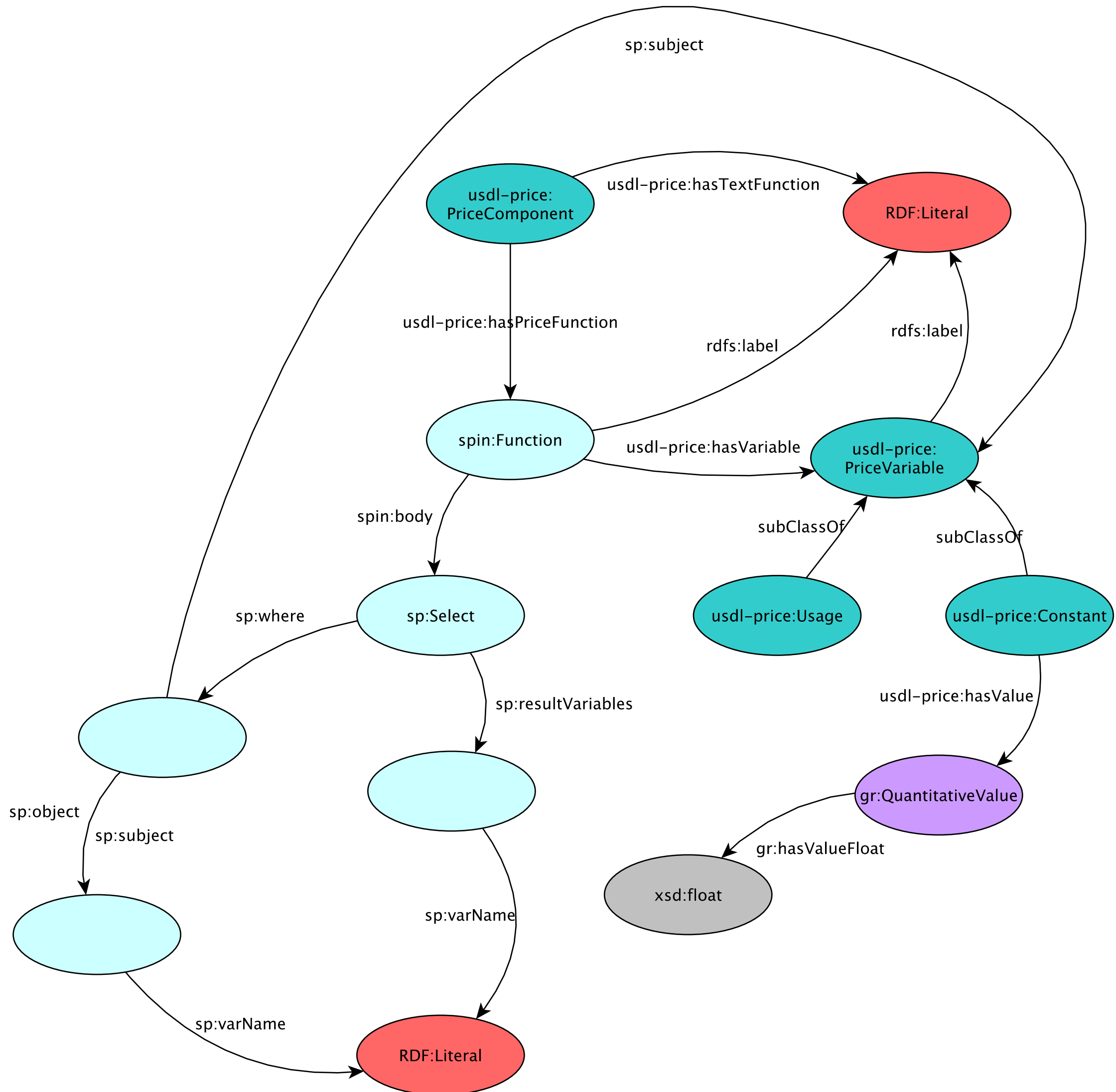


Figure 4.4: Price Function USDL model

4.2 Pricing Basis

The previous section describes the structure of the models used to represent digital assets and offerings within the FIWARE Business Framework. However, in the case of the pricing models, it is also needed to describe not only the structure, but also how these models are interpreted by the Store in order to calculate charges.

In this regard, the current section deals with this issue describing how the different pricing models are interpreted by the Store, providing some examples, in order to determine the way a concrete customer must be charged.

4.2.1 Price Plans

The Store allows providers to include multiple price plans in a single offering, where each price plan defines a different pricing model that can be selected by a customer. For example, a customer may select a price plan based on an initial payment and a monthly fee, while the offering also includes a pay-per-use model defined in another plan. When an offering includes multiple plans a *rdfs:label* property must be included since this field is used to identify the concrete plan. Additionally, the Store defines two special price plans, the updating price plan and a developers price plan. The updating price plan, which is identified by a *rdfs:label* “update” value, is used to define a price plan that only applies to customers that have acquired a previous version of the offering. On the other hand, the developers price plan, which is identified by a *rdfs:label* “developers” value, is a plan that can only be acquired by users with the developer role (have a look at section 3.3.1 for details on this topic) that typically use the acquired digital assets to develop new compositional applications. Note that it is not mandatory to include these special plans.

Following, it can be seen an example of a price plan node in turtle format.

```
1 <http://store.lab.fiware.org/ns#HQsRGctt8S1oH3jtB>
2   a usdl-price:PricePlan ;
3   dcterms:title "Example price plan"@en ;
4   dcterms:description "Price plan description"@en ;
5   rdfs:label "simple plan" ;
6   usdl-price:hasPriceComponent
7     <http://store.lab.fiware.org/ns#B4dyA6JWJpj3d>,
8     <http://store.lab.fiware.org/ns#xNUNpRqdWgokz> .
```

4.2.2 Pricing Models

As described in the Store architecture in section 3.3.1, the pricing models used by the Store are based on price parts that define simple models. These price parts can be combined arbitrarily to form flexible and complex pricing models for describing offerings. To represent the different price parts, the Store uses the *usdl-price:PriceComponent* class as defined in the Linked USDL specification.

The *gr:hasUnitOfMeasument* property of the *gr:PriceSpecification* class is used to identify what kind of pricing model is being used. For example, an offering that defines a pay-per-use service which also includes an initial payment can be represented using two price components. The first price component would represent the

initial payment and the other would define the pay-per use.

Depending on the value of the unit property, the price component can define the following pricing models:

- **Free offering**: If no price plan is defined the offering is considered to be free. Note that if only one of the digital assets offered is free, not specifying a price component for this asset is enough to ensure that the customer would not have a charge in this regard.
- **Single payment**: If the offering provider wants a single payment in her offering (i.e the offering offers a downloadable service such as a widget or an application, the offering offers a service that includes an initial charging, etc.) is enough to create a price component specifying the price and the currency (EUR, USD, etc.) and set the unit property to something like *single payment* (The value registered by default in the Store).
- **Subscription**: If the offering provider wants to specify some subscription models in her offering is enough to create a price component specifying the price, the currency, and the period of subscription setting the unit property to values such as *per day, per month, per year, per quarter.*
- **Pay per use**: If the offering provider wants to specify a pay-per-use model, the provider only needs to create a price component setting the unit property to the type of unit that would be monitored. Depending on the type of pay-per-use this property could have different types of values:
  - Pay per use event: *invocation, notification, transaction, session...*
  - Pay per use time: *second, hour, day, week...*
  - Pay per use quantity: *kilobyte, megabyte, gigabyte, data package, CPU instruction...*

Single Payment

The single payment is the most basic pricing model, which is supported by the Store. When providers include a price component with a unit that defines a single payment, they are including a charge that is made once on acquiring time, that is, customers are charged when they acquires the offering and then, the concrete price component has no effect in future charges.

Following, it can be seen an example of a price component defining a single payment.

```
1 <http://store.lab.fiware.org/ns#B4dyA6JWJpj3d>
2    a usdl-price:PriceComponent ;
3    rdfs:label "Single payment component"@en ;
4    dcterms:description "A single payment example"@en ;
5    usdl-price:hasPrice [
6       a gr:PriceSpecification ;
7       gr:hasCurrency "EUR" ;
8       gr:hasCurrencyValue "1"^^xsd:float ;
9       gr:hasUnitOfMeasurement "single payment"
10   ] .
```

Subscription

The second price model supported by the Store is the subscription-based model. When an offering includes a price component that defines a subscription, customers are charged on acquisition time, and then they are charged periodically depending on the renovation period defined by the included pricing unit. Note that the renovation period associated to a unit is defined by the Store admin that registered the unit.

Following, it can be seen an example of a price component defining a subscription.

```
1 <http://store.lab.fiware.org/ns#xNUNpRqdWgokzZzpm>
2    a usdl-price:PriceComponent ;
3    rdfs:label "Monthly subscription component"@en ;
4    dcterms:description "A subscription example"@en ;
5    usdl-price:hasPrice [
6       a gr:PriceSpecification ;
7       gr:hasCurrency "EUR" ;
8       gr:hasCurrencyValue "1"^^xsd:float ;
9       gr:hasUnitOfMeasurement "per month"
10   ] .
```

Pay-per-use

The most complex pricing model supported by the Store is the pay-per-use model. Unlike single payments and subscriptions that can only be defined by including a price component, the Store supports different ways of defining pay-per-use models. In this regard, for use-based models it is possible to define price components, price functions, and discounts.

Price Component Based

Defining pay-per-use models using price components is similar as defining subscriptions or single payments. The unit included in the price component define what is accounted, so the related charge is calculated by multiplying the consump-

tion made of the related unit for the defined value.

Following it can be seen an example of a price component defining a pay-per-use model.

```
1  <http://store.lab.fiware.org/ns#B4dyA6JWJpj3d>
2     a usdl-price:PriceComponent ;
3     rdfs:label "Pay-per-use component"@en ;
4     dcterms:description "A pay-per-use example"@en ;
5     usdl-price:hasPrice [
6        a gr:PriceSpecification ;
7        gr:hasCurrency "EUR" ;
8        gr:hasCurrencyValue "0.25"^^xsd:float ;
9        gr:hasUnitOfMeasurement "second"
10    ] .
```

Price Function Based

Typically, is possible to define pay-per-use models that depend on more than a single variable. For example an offering provider may want to define that the price of her service depends on the time the user access the service, but also on the megabytes downloaded. In this case, if the amount to be charged is calculated as the price of the consumption of time plus the price of the consumption of megabytes, it can achieved by using two price components since this is the default behaviour. However, if the service provider expects a more complex model, for example a model in which the final price depends a 40% on the time and a 60% on the megabytes, this cannot be done by using price components. To deal with this issue, the Store relies on the price functions.

Price functions are a way of specifying different kind of operations realized over usage variables and constants. Price functions can be included in a price component replacing the *gr:PriceSpecification* class; however, when a price function is included it is also needed to include a text function describing how the function works.

Following, it is possible to find a price component defining a price function.

```
1  <http://store.lab.fiware.org/ns#B4dyA6JWJpj3d>
2     a usdl-price:PriceComponent ;
3     rdfs:label "Pay-per-use price function"@en ;
4     dcterms:description "A pay-per-use example"@en ;
5     usdl-price:hasPriceFunction
6         <http://store.lab.fiware.org/ns#B4dyA6JWJpj3d> ;
7     usdl-price:hasTextFunction
8         "0.4*seconds usage + 0.6*megabytes usage" .
```

The first step to create a price function is including the different variables that are used in the function; these variables can be usage variables, which specify the

monitored variables, or constants.

```
<http://store.lab.fiware.org/second_usage>
   a usdl-price:Usage ;
   rdfs:label "seconds usage" .

<http://store.lab.fiware.org/megabyte_usage>
   a usdl-price:Usage ;
   rdfs:label "megabytes usage" .

<http://store.lab.fiware.org/constant40>
   a usdl-price:Constant ;
   rdfs:label "Constant 40" ;
   usdl-price:hasValue [
      a gr:QuantitativeValue ;
      gr:hasValueFloat "0.4"
   ] .

<http://store.lab.fiware.org/constant60>
   a usdl-price:Constant ;
   rdfs:label "Constant 60" ;
   usdl-price:hasValue [
      a gr:QuantitativeValue ;
      gr:hasValueFloat "0.6"
   ] .

<http://store.lab.fiware.org/pricefunct>
   a spin:Function ;
   rdfs:label "An example price function" ;
   usdl-price:hasVariable
      <http://store.lab.fiware.org/second_usage> ,
      <http://store.lab.fiware.org/megabyte_usage> ,
      <http://store.lab.fiware.org/constant40> ,
      <http://store.lab.fiware.org/constant60> ;

...
```

The function declaration itself is realized using the SPIN ontology that allows to include SPARQL queries inside RDF documents. In the case of the Store, this query should contain a list of tuples that are used to match the value of the declared variables with the auxiliary names that are used when evaluating the expression, and a BIND expression that defines how to calculate the price. This expression is declared as an operation between variables or other expressions. The allowed operations are Sum, Mul, Div, and Sub.

```
...
spin:body [
   a sp:Select ;
   sp:resultVariables ([ sp:varName "price"]) ;
   sp:where ([
      sp:subject <http://store.lab.fiware.org/constant40> ;
```

```
7          sp:predicate price:hasValue ;
8          sp:object [
9             sp:varName "caux1"
10         ]
11      ] [
12         sp:subject [
13            sp:varName "caux1"
14         ] ;
15         sp:predicate gr:hasValueFloat ;
16         sp:object [
17            sp:varName "40_perc"
18         ]
19      ] [
20         sp:subject <http://store.lab.fiware.org/constant60> ;
21         sp:predicate price:hasValue ;
22         sp:object [
23            sp:varName "caux2"
24         ]
25      ] [
26         sp:subject [
27            sp:varName "caux2"
28         ] ;
29         sp:predicate gr:hasValueFloat ;
30         sp:object [
31            sp:varName "60_perc"
32         ]
33      ] [
34         sp:subject <http://store.lab.fiware.org/second_usage> ;
35         sp:predicate price:hasValue ;
36         sp:object [
37            sp:varName "uaux1"
38         ]
39      ] [
40         sp:subject [
41            sp:varName "uaux1"
42         ] ;
43         sp:predicate gr:hasValueFloat ;
44         sp:object [
45            sp:varName "seconds"
46         ]
47      ] [
48         sp:subject <http://store.lab.fiware.org/megabyte_usage> ;
49         sp:predicate price:hasValue ;
50         sp:object [
51            sp:varName "uaux2"
52         ]
53      ] [
54         sp:subject [
55            sp:varName "uaux2"
56         ] ;
57         sp:predicate gr:hasValueFloat ;
58         sp:object [
59            sp:varName "megabytes"
```

```
60         ]
61     ] [
62         a sp:Bind ;
63         sp:expression [
64             a sp:Sum ;
65             sp:arg1 [
66                 a sp:Mul ;
67                 sp:arg1 [
68                     sp:varName "40_perc"
69                 ] ;
70                 sp:arg2 [
71                     sp:varName "seconds"
72                 ]
73             ] ;
74             sp:arg2 [
75                 a sp:Mul ;
76                 sp:arg1 [
77                     sp:varName "60_perc"
78                 ] ;
79                 sp:arg2 [
80                     sp:varName "megabytes"
81                 ]
82             ]
83         ] ;
84         sp:variable [
85             sp:varName "price"
86         ]
87     ])
88 ] .
```

The query included above is equivalent to:

```
1  SELECT ?price
2  WHERE {
3      <http://store.lab.fiware.org/constant40> price:hasValue ?caux1 .
4      ?caux1 gr:hasValueFloat ?40_perc .
5      <http://store.lab.fiware.org/constant60> price:hasValue ?caux2 .
6      ?caux2 gr:hasValueFloat ?60_perc .
7      <http://store.lab.fiware.org/second_usage> price:hasValue ?uaux1 .
8      ?uaux1 gr:hasValueFloat ?seconds .
9      <http://store.lab.fiware.org/megabyte_usage> price:hasValue ?uaux2 .
10     ?uaux2 gr:hasValueFloat ?megabytes .
11     BIND(((?seconds * ?40_perc) + (?megabytes * ?60_perc)) AS ?price) .
12 }
```

Discounts

An offering provider may want to define some kind of discounts depending on the usage of their digital assets, for example, consumption over a certain limit can has a 10% of discount. To achieve this, the Store uses discounts. Discounts are included in the price plan and have the same syntax as a payment so they can be defined with price components or price functions. When a discount is included, it

is calculated as a payment and then it is deducted from the final price.

```
<http://store.lab.fiware.org/ns#HQsRGctt8S1oH3jtB>
   a usdl-price:PricePlan ;
   dcterms:title "Pay-per-use Plan"@en ;
   rdfs:label "usage plan" ;
   dcterms:description "A price plan with a discount"@en ;
   usdl-price:hasPriceComponent
       <http://store.lab.fiware.org/ns#B4dyA6JWJpj3d> ,
       <http://store.lab.fiware.org/ns#gfs4A6JWJgt4> .

<http://store.lab.fiware.org/ns#gfs4A6JWJgt4>
   a usdl-price:Deduction ;
   rdfs:label "Deduction component"@en ;
   dcterms:description "A deduction example"@en ;
   usdl-price:hasPrice [
      a gr:PriceSpecification ;
      gr:hasCurrency "EUR" ;
      gr:hasCurrencyValue "0.01"^^xsd:float ;
      gr:hasUnitOfMeasurement "second"
   ] .
```

Chapter 5

Support for Multiple Digital Assets Support: The Plug-in model

One of the main objectives of the project (see section 1.3) is supporting any kind of digital assets in the FIWARE Business Framework. This chapter describes the solution that has been developed in order to achieve this objective.

As stated in section 1.1.3, the Store was able to manage two different types of resources. On the hand, the store dealt with services and other resources that were accessible thought an URL using *API resources*. Those resources were registered in the Store just giving their basic information and providing the URL where they were deployed. On the other hand, the store also allowed providers to monetize resources that were available as a file using *Downloadable resources*. In this case, providers could upload their resource directly to the Store, which controlled the access and managed the downloads.

It is important to remark, that this model allowed providers to include almost any kind of digital asset in the Store, since most of the digital assets can be described as an API or included in a file. However, this model was not enough for giving providers a real monetization service for several reasons:

- The two types of resources managed by the Store were too general. In this regard, customers cannot really know exactly what they were acquiring.
- Some specific types of resources may require customers to provide some information that cannot be retrieved without knowing exactly what the resource is.
- Some resources might need to be validated; this cannot be achieved without knowing the exact format of the digital asset.

- There are some scenarios where it may be required to extract some information from the resource itself.
- For resources representing a backend service, it might be necessary to validate that the provider user is allowed to register the concrete resource. This action cannot be achieved without knowing the structure of the APIs provided by the resource.

These problems might point out that for this kind of monetization framework it is required to have built-in specific types of assets, instead of general ones. However, this approach lacks of flexibility and force to modify the software of the different components every time that it is required to include a new type of digital asset.

To deal with these problems, it has been decided to follow an intermediate approach that allows to exploit the benefits of not limiting the types of assets supported by the framework, but without creating a too general model. Concretely, its has been decided to create a plug-in based model. In this way, the Store works with specific types of resources, so it is possible to retrieve specific information, validate the resource, etc., but when a new type of asset is required, it can be loaded as a plug-in without the need of modifying the source code of any of the FIWARE Business Framework components.

5.1 Plug-in Model

As stated in section 3.3.1, the Store in the component where the digital assets are registered (or uploaded) as resources. In this regard, is in the Store where the system for the loading and management of plug-ins has been created.

Figure 5.1 shows the basic architecture of the plug-in management system that have been included in the Store. This system has been divided into several modules whose objective is abstracting the existing plug-ins to the rest of the component. Concretely, the following modules has been included:

- **Plug-in administration interface**: The objective of this module is to allow administrators of the Store instance to manage the different plug-ins. To do that, this module allows providers to load, list, and remove plug-ins.
- **Plug-in manager**: This module is the core of the plug-in management system, and is in charge of two main tasks. On the one hand, the plug-in manager executes the tasks received though the plug-in interface, so this module builds and remove the needed data structures and loads the plug-in to the plug-in container. On the other hand, this module is in charge of managing the different events generated during the management of resources in the Store, choosing the correct plug-in and event handler.

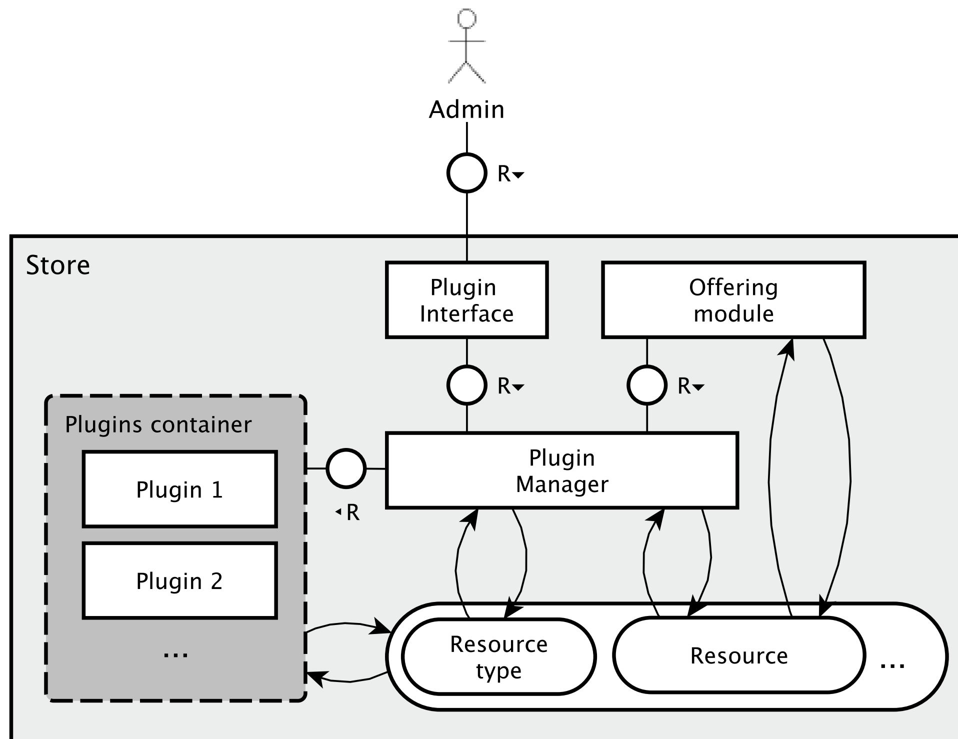


Figure 5.1: Plug-in Model Architecture

- **Plug-in container**: The objective of this module is storing the plug-ins loaded by the system administrator.

Apart from the new modules included in the Store architecture, it has also been needed to include new data models in order to support the plug-in based approach. As stated in section 3.3.1, *Resource model* has a *Resource type* field which specifies what the resource is (API or Downloadable). For compatibility reasons, it has been decided to use this field to identify the type of digital asset in the plug-in based approach, that is, the resource type defined by the plug-in.

In addition, it has been included a new data model, called *Resource type*, to the Store model. This data model describes all the information that is required by the Store, including basic, technical, and configuration information. As shown in figure 5.1, the *Resource type* data model is only accessed by the plug-in manager and by the existing plug-ins. The *Resource type* model contains the following fields:

- **Id**: This field automatically generated and is used to identify the plug-in.

- **Defined type**: This field defines the type of digital asset that is managed by the concrete plug-in (e.g CKAN Dataset, Context Broker Stream, etc.).

- **Author**: This field contains the author that developed the plug-in.
- **Version**: This field contains the version of the plug-in.
- **Media types**: This field specifies what are the valid media types (e.g application/zip, text/csv, etc.) when creating a resource whose type is the one defined by the plug-in.
- **Formats**: This field specifies what are the valid formats for providing a resource of the defined type. It is needed to take into account that the Store allows to provide resources in two different ways, uploading a file with the resource or proving an URL. Those are the formats that are specified in this field (file vs URL).
- **Overrides**: This field describes whether the plug-in will programmatically override the value of any of the fields of the created resources.
- **Form**: This field contains the specification of a form that will be used to retrieve arbitrary meta data required by the concrete plug-in.
- **Module**: This field is used for technical purposes and allows to locate the source code of the plug-in that is used to process the different events generated by the Store.

The main objective of having a plug-in based model (apart from the one of having specific types of resources), is being able to include programmatic validations and controls, to override data, to access the content of the resource, etc. To do that, it has been created some events that allow the creator of the plug-in to to deal with the stated objectives by creating a couple of event handlers. In this regard, some events have been created for the different actions that can be performed regarding resources.

For the creation of a new resource the following events have been created:

- **on pre create validation**: This event is raised before the Store validates the information of the resource given by the provider. The main objective of this event is allowing the concrete plug-in to override some information of the resource, which is then validated by the Store. The handler of this event receives the raw information given by the provider for the creation of the resource.
- **on post create validation**: This event is raised after the information given by the provider for creating the resource has been validated by the Store. The main objective of this event is allowing plug-ins to override resource information that is not intended to be validated by the Store. The event handler of this event receives the validated information given by the provider.

- **on pre create**: This event is raised before a new resource is saved to the database. This event can be used to modify the resource object before saving it or for executing some tasks required by the concrete type of resource, such as generating some meta data to be saved with resource or performing specific validations. The handler of this event is called passing the resource object as a parameter.
- **on post create**: This event is raised after a new resource has been saved to the database. The intention of this event is allowing the plug-in to perform some tasks that depend on the complete creation of the resource, for example notifying a server that a resource has been created in case it might be necessary. The handler of this event receives the saved resource object.

For the updating of a resource the following events have been created:

- **on pre update**: This event is raised before the Store saves the result of updating the basic info of a resource. This event is intended to allow plug-ins to perform specific validations of the data or override some fields. The handler of this event receives the modified resource object before saving it.
- **on post update**: This event is raised after the Store has saved the result of updating the basic info of a resource. The main objective of this event is allowing plug-ins to execute specific tasks that require the updated resource to have been saved in the database (e.g sending notifications). The handler of this event receives the modified resource object already saved in the database.

For the upgrading of a resource the following events have been created:

- **on pre upgrade validation**: This event is raised before the Store validates the information of a new version of a resource given by the provider. The main objective of this event is allowing the concrete plug-in to override some information of the new version of the resource, which is then validated by the Store. The handler of this event receives the raw information given by the provider for upgrading of the resource and the resource object.
- **on post upgrade validation**: This event is raised after the information given by the provider for upgrading the resource has been validated by the Store. The main objective of this event is allowing plug-ins to override some information of the new version of the resource that is not intended to be validated by the Store. The event handler of this event receives the validated information given by the provider and the resource object.
- **on pre upgrade**: This event is raised before a new version of a resource is saved to the database. This event can be used to modify the resource object before saving it or for executing some tasks required by the concrete type of resource, such as generating some meta data to be saved with resource or performing specific validations. The handler of this event is called passing the resource object as a parameter.

- **on post upgrade**: This event is raised after a new version of a resource has been saved to the database. The intention of this event is allowing the plug-in to perform some tasks that depend on the upgrade of the resource, for example notifying a server that a resource has been upgraded. The handler of this event receives the saved resource object.

For the deletion of resources the following events have been created:

- **on pre delete**: This event is raised before the Store deletes a resource. This event is intended to allow plug-ins to perform specific tasks and validations before a resource is removed. In this regard, a concrete resource type might require some actions to have been tackled by the provider of the resource before allowing a deletion. The handler of this event receives the resource object to be deleted.
- **on post delete**: This event is raised after the Store has deleted a resource. The main objective of this event is allowing plug-ins to execute specific tasks that require the resource to have been deleted from the database (e.g sending notifications). The handler of this event receives a copy of the deleted resource.

Finally, it has also been defined some events related with the interaction with offerings that include resources of a given resource type.

- **on pre binding**: This event is raised before a resource is bound to a given offering. This event is intended to allow providers to check specific permissions or make specific validations before a resource is included in a new offering. The handler of this event receives, the resource object and the offering object, as well as the configuration information provided by the user (mainly the associated price component).
- **on post binding**: This event is raised after a resource has been bound to an offering. This event allows providers to manage the binding of their resource, for example making notifications. The handler of this event receives, the resource object and the offering object, as well as the configuration information provided by the user.
- **on pre acquire**: This event is raised before the acquisition process of an offering which includes the related resource is finished. The main objective of this event is allowing providers to perform specific resource validations before allowing a customer to acquire the given resource. The handler of this event receives the resource object and the purchase object pending to be confirmed.
- **on post acquire**: This event is raised after an offering which includes a given resource has been acquired. This event is intended to allow providers to manage the acquisition of their resources, for example this event can be used for notifying providers that of one of their services has been acquired. The handler of this event receives the resource, and the purchase objects.

It is needed to take into account that its has been supposed that creators of the plug-ins are also administrators of the concrete Store instance, or al least trusted developers. In this way, the source code of the plug-in is executed as the Store code is, so there are no special access restrictions applying to resource plug-ins.

The events allow to modify some aspects of the management of resources within the concrete Store instance. Additionally, its has also been included the possibility of providing configuration options that modify some default behaviours of the Store, regarding the creation and modification of resources. This configuration options has been introduced with the description of the *Resource type* data model. First of all, the plug-in model allows to limit the supported media types. By default, the Store allows providers to include any media type when creating a resource; however, it is possible to define a specific set of media types for a given resource type, so no other media type can be chosen.

Moreover, it is possible to specify how the resource itself is provided for a given resource type. By default, the Store allows to provide both, a URL or a file for creating a resource. Nevertheless, it is possible to configure a concrete resource type in order to choose the way the resource have to be provided. This is useful for example when defining a type of resource which is a backend service, thus it does not make sense to register it uploading a file.

In addition, there are some cases when it make sense to override (or automatically fill) the information provided by the creator of the resource, for example when the resource is a packaged app which contains its name, version, etc. within the file uploaded to the Store, or when it is possible to automatically determine the media type from the resource. In these cases, it is possible to configure the resource type specifying those fields that will be overridden in order to make the Store aware of this.

The different types of resources might require different information. To deal with this, it is possible to configure a resource type specifying a series of form fields. Those fields are used to create a form that is displayed to the provider when creating a new resource, and allow to retrieve the extra information regarding the resource, which is then stored as meta data.

Finally, it is important to notice that this plug-in model is integrated with the Linked USDL document generation for resources (see section 4). In this regard, when a new resource is created, and thus its Linked USDL description is generated, the resource type is included in this description as the label of the *skos:Concept* class. In this way, the resource type defined in the Store is available to other components of the FIWARE Business Framework, being used for example for classification of offerings in the Marketplace (see section 8).

5.2 Creating a Plug-in

Section 5.1 specifies the basic architecture and model that has been created for supporting a plug-in based approach for the management of resources. This section explains in detail how to create a plug-in and how to apply the different concepts introduced in the previous section.

Plug-ins are managed by the Store as a package. This package is a zip file that can have any internal structure, but taking into account that a file *package.json* is required in the root of the file structure. This file will be used by the store in order to determine the basic information of the plug-in and the selected configuration options.

Following its is possible to find a minimal *package.json* file:

```
1  {
2    "defined_type": "CKAN Dataset",
3    "author": "fdelavega",
4    "version": "1.0",
5    "module": "ckan_dataset.CKANDataset",
6    "media_types": [],
7    "formats": ["FILE", "URL"],
8    "overrides": [],
9    "form": {}
10 }
```

It can be seen that the *package.json* file contains two groups of parameters. First of all, it includes some basic fields containing the information of the resource, which describe the defined resource type, and the author and version of the plug-in. On the other hand, the *package.json* file also includes the needed fields that are used for the configuration of the plug-in.

As stated in section 5.1, the plug-in model defines some events that can be managed. To create software which is able to handle the existing events the Store defines an abstract class (*Plugin* class), which contains a method for every of the existing events. In this regard, the main class of the plug-in must be a subclass of the *Plugin* class and implement the methods for the different events. Note, that the Store software is written in Python, so it is required to use this language for developing a plug-in. In order to locate the main class of the plug-in, the file *package.json* must include a *module* parameter. This parameter contains the relative path (in Python format) of the related main class.

Following it can be seen the basic structure of a plug-in class:

```

from wstore.offerings.resource_plugin.plugin import Plugin

class CKANDataset(Plugin):

    def on_pre_create_validation(self, provider, data, file=None):
        pass

    def on_post_create(self, resource):
        pass

...
```

As stated before, it is also possible to define some specific configuration parameters. In this regard, *media_type* parameter allows to define what are the valid media types for the concrete resource type by providing a list of them. It is important to remark that providing an empty list makes the Store accept any media type for the resource type.

Additionally, the plug-in model also allows to define the valid format for providing the resource. In this way, it is possible to specify those formats using the *formats* parameter in *package.json*. This parameter is a list that can contain the values FILE and URL, specifying that the resource can be provided uploading a file or giving a URL respectively.

Figure 5.2: Form for providing a resource showing media type and format selection

Moreover, in case the resource plug-in overrides any value given by the provider when creating a new resource, it is possible to specify what fields are going to be overridden by the plug-in in order to make the Store aware of this. Note, that notifying the Store what fields are going to be overridden, allows it to change its default behaviour and interface (e.g not asking the provider to include values that are going to be overridden). To specify, what fields are overridden the plug-in, the

file *package.json* includes the parameter *overrides*. This parameter is a list that can contain any of the following values: NAME, VERSION, OPEN, DESCRIPTION, MEDIA.

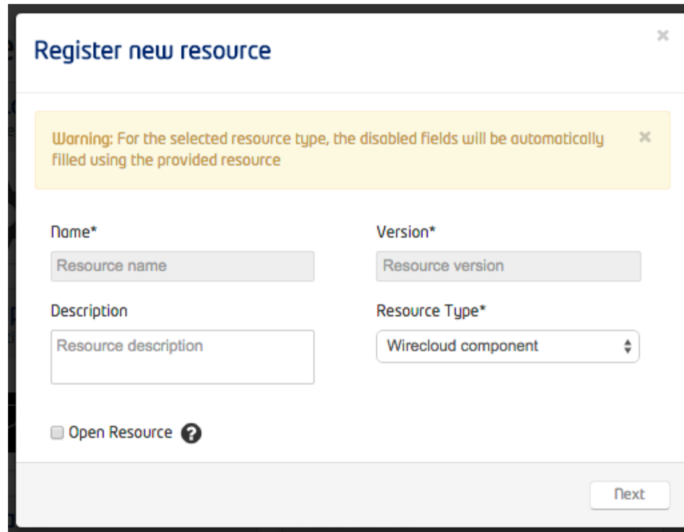


Figure 5.3: Form for providing basic resource info with overridden fields

To end with the configuration parameters, it is also possible to define a form for retrieving specific meta data. To do that, the field *form* of the *package.json* file is used. This field is an object that allows to declare different input fields and ids, that are then rendered by the Store in order to build the specified form.

An example of *package.json* file defining a form follows:

```
{
  "defined_type": "CKAN Dataset",
  "author": "fdelavega",
  "version": "1.0",
  "module": "ckan_dataset.CKANDataset",
  "media_types": [],
  "formats": ["FILE", "URL"],
  "overrides": [],
  "form": {
      "notif": {
         "type": "text",
         "placeholder": "Notification URL",
         "default": "http://data.lab.fiware.org/notify_creation",
         "label": "Notification URL",
         "mandatory": true
      },
      "license" : {
          "type": "select",
          "label": "Dataset license",
```

```
20             "options": [{
21                 "text": "Creative Commons",
22                 "value": "opt1"
23             }, {
24                 "text": "BSD",
25                 "value": "opt2"
26             }]
27         },
28         "is_private": {
29             "type": "checkbox",
30             "label": "Is private",
31             "text": "Check if the provided dataset is private or not",
32             "default": true
33         },
34         "add_data": {
35             "type": "textarea",
36             "label": "Additional data",
37             "placeholder": "Additional data"
38         }
39     }
40 }
```

It can be seen in the example above that keys of the dictionary contain the id of the concrete field. This value is used, on the one hand, as the id of the HTML input field in the rendered form, and on the other hand, as the key when saving the retrieved information as meta data of the resource. Moreover, the example shows the basic structure for defining an input field. First of all, all the input fields contain a *type* parameter which defines the type of input, and whose value correspond to the existing types of inputs in HTML. Note that only text, select, checkbox, and textarea are supported. Additionally, inputs contain a series of parameters that correspond with existing attributes in HTML for the concrete type of form input (e.g placeholder for text input). Finally, it has also been included an additional parameter (*mandatory*) which can be included in any of the inputs, and specifies that the provider must give a value for this input when creating a new resource. Note that all input fields are not mandatory by default. Figure 5.4 shows the result of rendering the form included in the example.

Once the plug-in has been created, it can be managed in the Store thought the plug-in interface defined in figure 5.1. In practise, it is done using some management commands that have been develop as part of the Store.

Register new resource

Notification URL

http://data.lab.fiware.org/dataset?act

Additional data

Additional data

Dataset license

Creative Commons

Is private

Check if the provided dataset is private or not

Accept

Figure 5.4: Autogenerated form for retrieving resource meta info

Chapter 6

Pay-Per-Use: The Accounting Proxy

As stated in section 1.3, one of the main objectives of the FIWARE Business Framework is supporting pay-per-use pricing models. To deal with this objective, it is necessary to take into account two different aspects that must be covered. On the one hand, the FIWARE Business Framework must be able to support the pay-per-use in its offering models, and being able to charge customers based on them. On the other hand, it is required to have a component able to perform the accounting of the offered services, in order to retrieve the needed usage information. Having in mind the existing requirements, the current chapter explains the found solution to deal with pay-per-use and accounting in the FIWARE Business Framework.

Chapter 4 describes how pay-per-use pricing models are serialized in Linked USDL format. This RDF model will be used as the starting point of this chapter in order to describe how pay-per-use and accounting are effectively managed.

6.1 Store Pay-Per-Use Management

Within the FIWARE Business Framework architecture, the Store is the component in charge of managing offerings and charging customers. In this regard, the Store is the component that is able to interpret the pricing models and that require usage information in order to charge for pay-per-use services. To deal with these tasks, the Store parses the pay-per-use pricing models as defined in chapter 4 and generates an internal structure that allows to calculate the amount to be charged to a concrete customer, using the accounting information.

Figure 6.1 shows the typical structure used in the Store for charging pay-per-use services. In this approach, the Store relies in an external component whose objective is monitoring those services, offered under a pay-per-use model, and generating accounting information. The accounting information is then feed to the Store as SDR (*Service Detailed Record*) documents.

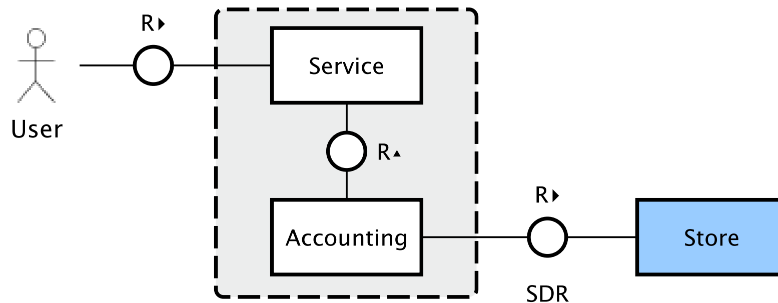


Figure 6.1: Basic Accounting Model

As stated in chapter 4 pay-per-use models are managed in the Store in two different ways, using *price components*, and using *price functions* with *price variables*. At this point, it is necessary to introduce the concept of *monitored variable*. A *Monitored variable* is the concrete unit being monitored, for a concrete service, using an accounting component. In addition, *monitored variables* are referenced in the pay-per-use model in different ways, depending on how the usage-based charging is calculated. If the price model is based in price components, every component is describing a monitored variable. On the other hand, if the price model is based in price functions, monitored variables are described using price variables. In both cases, the monitored variable is identified using the *rdfs:label* property of the concrete node.

The following example shows the same monitored variable, described using a price component and a price variable.

```
1  <http://store.lab.fiware.org/ns#B4dyA6JWJpj3d>
2     a price:PriceComponent ;
3     dcterms:title "Seconds usage price component"@en ;
4     rdfs:label "seconds usage" ;
5     dcterms:description "A pay-per-use example"@en ;
6     price:hasPrice [
7        a gr:PriceSpecification ;
8        gr:hasCurrency "EUR" ;
9        gr:hasCurrencyValue "0.25"^^xsd:float ;
10       gr:hasUnitOfMeasurement "second" .
11
12 ...
13
14 <http://store.lab.fiware.org/variable_usage>
15    a price:Usage ;
16    rdfs:label "seconds usage" .
```

As stated before, the Store requires accounting information to be able to calculate pay-per-use charges. In order to retrieve this information, the Store offers a push-oriented API to gather SDR documents generated by the accounting component. The following example contains a SDR document providing accounting information for the monitored variable described in the previous example.

```
{
  "offering": {
    "organization": "CoNWeT",
    "name": "SensorDataStream",
    "version": "1.0"
  },
  "component_label": "seconds usage"
  "customer": "fdelavega",
  "time_stamp": "2014-02-21 10:00:00.0",
  "correlation_number": "105",
  "record_type": "time",
  "unit": "second",
  "value": "60"
}
```

In the example above, it can be seen the different fields that must be included in a SDR document. Concretely, a SDR document must include the following fields:

- **offering**: This field is used to identify the offering that have been acquired and whose pricing model is used. Note that the same service (resource) can be included in several offerings, so different pricing models can be applied.
- **component_label**: This field identifies the monitored variable, and is used to match the accounting information with a concrete price component or variable.
- **customer**: This field contains the id of the user that generates the accounting. It is necessary to remark, that this user does not need to be the same who acquired the offering. As stated in section 3.3.1, the Store is able to manage organizations, so multiple users can have been granted to access a service in a given acquisition.
- **time_stamp**: This field contains the time stamp created when the accounting was generated.
- **correlation_number**: This field contains a correlation number, which is incremented in every SDR, and is used to ensure that no accounting information has been lost.
- **record_type**: This field is used to specify the type of pay-per-use that has been monitored (time, event, or quantity).
- **unit**: This field specifies the concrete unit that has been monitored (e.g second, megabyte, etc)

- **value**: This field contains the actual usage that has been made since the last SDR was sent to the Store.

Once the Store has received accounting information from an accounting component, it is able to calculate the charging for the given customer. This process, can start in two different ways: (1) If the pricing model of the given offering includes a subscription price part, the aggregation of accounting information and the charging calculus of the pay-per-use parts is launched every time that the subscription is renovated. (2) If the pricing model of the given offering only includes pay-per-use and single payments parts, the aggregation of accounting information and the charging calculus is launched periodically, depending of the configuration of the concrete Store instance.

Once the pay-per-use charging process has been launched, the first step is aggregating the accounting information. In this regard, the store aggregates all the usage information received from the accounting component, since the last aggregation period, grouped by monitored variable in order to determine the total usage during the period. Then, the Store uses the total usage information to calculate the final amount to be charged using the pricing model of the given offering. This calculus depends on the type of pay-per-use model defined: (1) If the monitored variables are mapped with pricing components, the total amount to be charged is calculated multiplying the total usage by the per unit value included within the price component (*gr:hasCurrencyValue*). (2) If the monitored variables are mapped with price variables associated with a price function, the total amount to be charged is calculated by applying the concrete price function to the total usage values.

6.2 Accounting Proxy

The section 6.1 shows how the Store calculates charges using pricing models and accounting information. At this point, it is necessary to explain how the accounting information is generated, and how pay-per-use services are handled. In this regard, a support component of the FIWARE Business Frameowork has been created, the Accounting Proxy. This component has been designed in order to provide two main features. On the one hand, it checks if a given user is authorized to access a concrete service (the user has acquired an offering that contains the related resource) and control the access. On the other hand, the Accounting proxy intercepts requests to the monitored service in order to create the accounting information. The current section describes the architecture of the Accounting Proxy and its integration within the FIWARE Business Framework in order to manage pay-per-use services.

6.2.1 Accounting Proxy Architecture

In order to make the Accounting Proxy as much flexible as possible, it has been designed to be able to monitor an arbitrary number of services. It this approach, services do not need to be separate systems, but they could be different paths within the API of the same system. This allow, not only to make the accounting of different backend services, but also to register the usage of resources which are accessible via API inside a bigger system, e.g a dataset offered as an API which is included in an open data management system.

To be able to account different services, the Accounting Proxy allows to register them using an approach based on internal an external URLs. To do that, an administrator of the Accounting Proxy (and of the monitored services) has to provide the local URL of the service including the port and the path. Moreover, the administrator has also to provide a public path that will be then used to access the service. It is necessary to remark, that the local URL of the registered service must not be accessible from the outside of the existing environment, otherwise users would be able to directly access the service without using the public path specified in the Accounting Proxy, making impossible to monitor the usage.

Figure 6.2 shows an example on how local and external URLs work. In can be seen that in this example there are three monitored services running in the same server: two of them running over a tomcat in the URLs *http://localhost:8080/service1* and *http://localhost:8080/service2*, and another service running in *http://localhost:5956/*. Moreover, the Accounting Proxy is deployed using a public host, which is the one used by the different users in order to access the services.

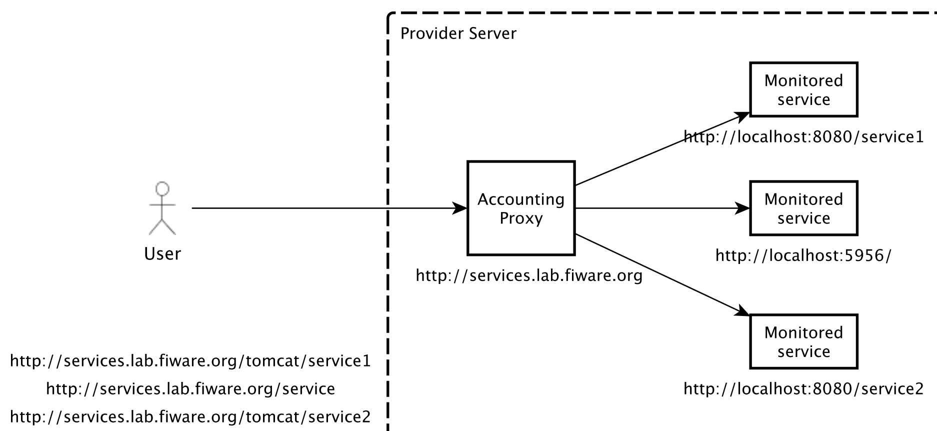


Figure 6.2: Monitored Services URL Management

In order to the register the monitored services included in figure 6.2, an administrator of the system had to provide the internal URLs of the different services

(localhost) and include a path where the services would be available, in this case */tomcat/service1*, */tomcat/service2* and */service*. Once the different services has been registered, they become available through the host of the accounting proxy and the path provided by the system administrator. Therefore, in this case the services can be accessed from the URLs *http://services.lab.fiware.org/tomcat/service1*, *http://services.lab.fiware.org/tomcat/service2* and *http://services.lab.fiware.org/service*. Note that in this example, the monitored services are deployed in the same server as the Accounting Proxy, using localhost in order to avoid users to access those services without going through the proxy. However, this is not mandatory, monitored services can be deployed in different servers as the Accounting Proxy if the provider ensures that only the proxy can access them (e.g using firewalls).

Figure 6.3 shows the internal architecture of the Accounting Proxy. It can be seen that users, who use the created external URLs for accessing the monitored services, sent their requests to the *Access Interface*. The objective of this module is receiving the external requests and abstracting the internal structure to the existing users by sending them the response received from the monitored service.

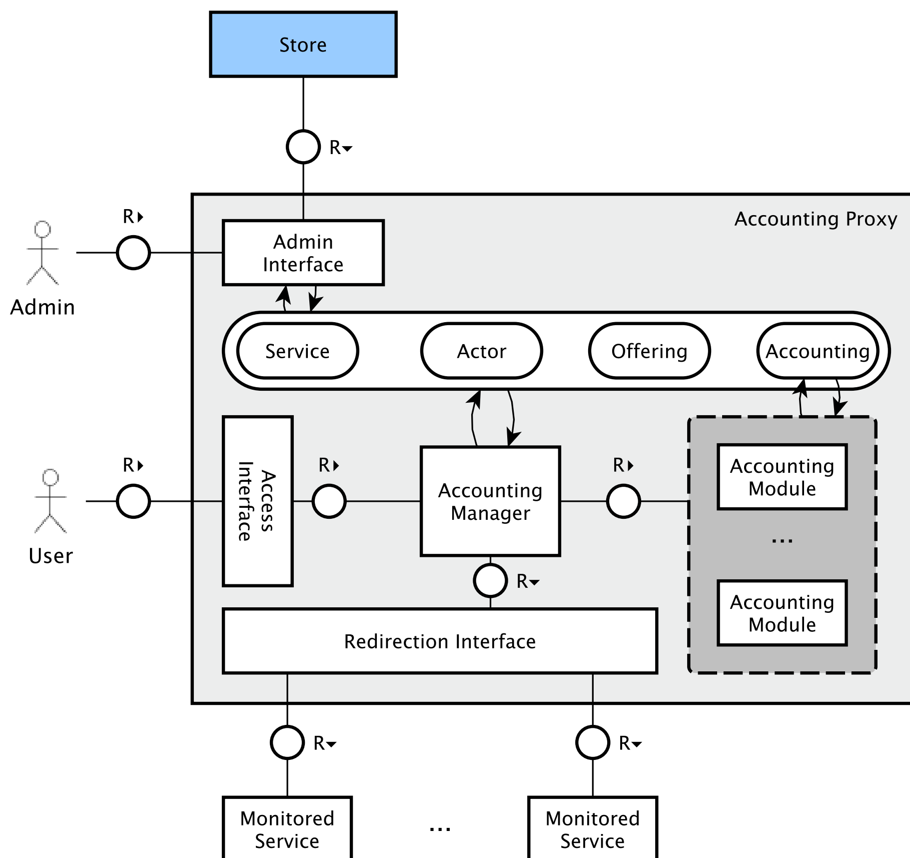


Figure 6.3: Accounting Proxy Architecture

Once the *Access Interface* receives a request from an external user, it passes the request to the *Accounting Manager*, which is in charge of supervising the accounting process. Then, the *Accounting Manager* uses the information stored in the database and the accessed path to determine the endpoint of the monitored service. In the next step, the *Accounting Manager* passes the endpoint of the monitored service and the request to the *Redirection Interface* module, which sends the user request to the monitored service and receives the response. Finally, the response received from the monitored service is sent back to the *Access Interface* (through the *Accounting Manager*) which sends it to the user.

During the redirection process described above, the Accounting Proxy needs to monitor the usage of the service in order to save accounting information. To deal with this task, the *Accounting Manager* make use of the *Accounting Modules*. Concretely, it determines the concrete unit to be monitored extracting this information from the database (section 6.2.2 explains in detail how units are bound to services) and calls the related *Accounting Module* with the request and response information provided by the user and the monitored service respectively.

As can be extracted from the previous paragraph, the Accounting Proxy is able to support different units for monitoring services (e.g calls, megabytes, seconds, etc.). In this way, the Accounting Proxy relies in an *Accounting Module* for monitoring the usage of the services according to a given unit. To perform this action, an *Accounting Module* receives the request made by the user and the response returned by the monitored service, and uses this information to calculate the effective usage within the call. Next, the calculated usage is aggregated and saved with the existing accounting information for the given service and user. For example, an *Accounting Module* in charge of monitoring a megabyte-based accounting, would calculate the size of the the response returned by the service and then add this quantity to the total amount consumed by the user for this service. This approach, allows to extend the Accounting Proxy any time a new accounting unit is needed, providing flexibility and giving providers the possibility of creating their own *Accounting Modules* when they require a different unit to be monitored.

Apart from the modules intended for user access, the Accounting Proxy also provides a module intended to be used for administration purposes (*Administration Interface*), which can be accessed from different sources. On the one hand, this module is used by administrators of the proxy for the management of monitored services, including the registration and deletion of them. On the other hand, this module is used by the Store for feeding offering specific information that is required during the accounting process.

The Accounting Proxy has been developed to feed the Store (or an external system which implements the required APIs) with accounting information, thus, monitored services are offered as resources in the Store. As stated in section 3.3.1,

the same resource can be included in several offerings, so the same resource can be offered under different pricing models. That means that it might be required to use different accounting modules depending on who is the user accessing the service, and what are the concrete pricing that applies in this access.

To deal with the issue of having multiple pay-per-use models applying to the same monitored service, the Accounting Proxy offers an API that must be invoked every time the monitored service is included in a new offering. In this way, the Accounting Proxy saves information about what offerings the service is included in, and what is the unit to be monitored for each of them. Following, it can be seen an example of the expected information that must be included in this notification.

```
{
  "offering": {
      "organization": "fdelavega",
      "name": "RealTimeStream",
      "version": "1.0"
  },
  "url": "http://services.lab.fiware.org/service",
  "record_type": "quantity",
  "component_label": "megabyte component",
  "unit": "megabyte"
}
```

The example above shows that the notification must include fields for two different purposes. On the one hand, it contains fields for identifying the offering and the service it is referring to. On the other hand, it also contains a couple of fields describing the concrete monitored variable. Specifically, the notification must contain the following fields:

- **offering**: This field contains the identification of the offering where the service has been included.
- **url**: This field contains the URL which has been used to create the resource. Note that this URL is the external URL of the monitored service.
- **record_type**: This field specifies the type of pay-per-use (quantity, time, event) which is specified in the concrete component label.
- **component_label**: This field identifies the price component (or price variable) that will be applied to the accounting information generated for the specified service.
- **unit**: This field contains the concrete unit to be monitored for the given service.

Using the notification described in the previous example, the Accounting Proxy is able to determine what are the different possibilities when generating the accounting of a given service. However, the Accounting Proxy also needs to know what are

the offerings acquired by a concrete customer, when she accesses a service, in order to determine what is the concrete unit to be applied.

Again, the Accounting Proxy deals with this problem by exposing an administration API, where the Store can feed acquisition notifications. In this way, the Accounting Proxy is notified when a customer acquires an offering which contains a related monitored service. Note, that this information is not only used to determine the unit, but also to authorize the accesses to existing service since the Accounting Proxy actually know is the customer has acquired a given service or not. Following, it can be seen an example of an acquisition notification.


```
1  {
2    "offering": {
3        "organization": "fdelavega",
4        "name": "RealTimeStream",
5        "version": "1.0"
6    },
7    "customer": "conwet",
8    "reference": "111222333",
9    "resources": []
10 }
```


It is important to remark, that this notification acquisition contains the same fields as described in the Store architecture in section 3.3.1.

With the information described at this point, the Accounting Proxy would be able to calculate the accounting for the usage made by the different customers, if they could not acquire two different offerings where the same resource has been included. However, this is not the case, the Store does not limit customers when acquiring offerings, so they are able to acquire an offering which includes a resource they have already acquired if they feel the new pricing is better for them. That means that a customer might have several offerings containing the same monitored service with different units.

Additionally, it may also be possible that a customer has acquired a given resource which has also been acquired by an organization she belongs. In this case, the customer may want to choose whether she accesses the service on behalf the organization or on behalf herself.

The Accounting Proxy solves both problems by forcing users to specify the context of the access in the different requests. This is done by including a header with an *API Key*, which is automatically generated by the proxy when an acquisition notification is received. This *API Key* specifies the context of the access by identifying both, the customer and the offering. Note that the customer identified by the *API Key* is not necessarily an user, but it can also be an organization if the offering has been acquired for it.

When customers access a monitored service, their requests will include a *X-API-Key* header with an API Key. Once the Accounting Proxy receives a request, it determines which is the offering and the customer identified by the *API Key*, and checks whether the user accessing the service in the one who acquired the offering, or belongs to the organization which acquired it, depending on the case. Then, it uses the information saved when the binding notification was sent in order to determine the unit that must be accounted in the concrete context.

Additionally, it is needed to remark that the Accounting Proxy sends all the information contained in its database to the Store periodically, in order to feed it with accounting information. To do that, it generates SDR documents (as described in section 6.1) for each of the monitored services. Note, that the Accounting proxy is able to create SDRs since the information required to be included in these documents is sent to the Accounting Proxy in the binding notifications.

Finally, it is needed to take into account that the Accounting proxy is designed to work with a single instance of the Store which is registered including its URL. In this regard, the Accounting Proxy uses the URL included by configuration in order to send the accounting information.

6.2.2 Integration with the Framework

Section 6.2.1 shows the basic architecture and functionality provided by the Accounting Proxy. However, this component needs to be integrated within the FIWARE Business Framework ecosystem. In this respect, this section covers the work that have been carried out in order to integrate the Accounting Proxy.

First of all, the Accounting Proxy has authorization features, since it determines if users can access a given service depending on if they have acquired or not an offering containing it. However, the Accounting Proxy does not offer authentication features. To deal with this issue the Accounting Proxy relies in the Identity Manager (as the rest of the components of the framework do) and in an small component given with it called *PEP Proxy* (Have a look at annex B.1 for details on this software). Figure 6.4 shows the process of accessing a monitored service including authentication of users.

Following it can be found the different steps included in figure 6.4

1. The user uses the external URL of the monitored service and makes a request including an access token (see section 3.2 for details on how this token is obtained) and the API Key which identifies the context of the access.

2. The PEP Proxy intercepts the request and authenticates the access token in the Identity Manager.

3. The Identity Manager returns information of the concrete user, identified by the access token, to the PEP Proxy, including basic information of the user such as the id and the organizations she belongs.

4. The PEP Proxy sends the request of the user to the Accounting Proxy, including as headers all the information of the user given by the Identity Manager.

5. The Accounting Proxy accesses the monitored service. Note that before this step, the Accounting Proxy validates the API Key using its stored information and the user id (or organization id) provided by the PEP Proxy.

6. The monitored service sends a response to the Accounting Proxy.

7. The Accounting Proxy sends back the response to the PEP Proxy

8. The PEP Proxy sends the response to the user.

Note that in practise the PEP Proxy and the Accounting Proxy can be collapsed in a single system, in order to avoid unnecessary redirections.

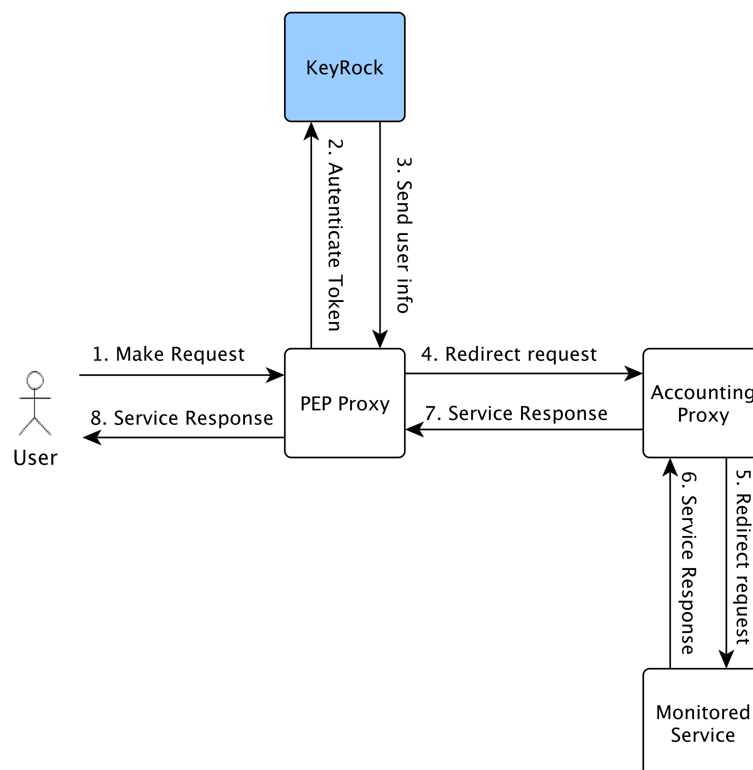


Figure 6.4: Service access process through the Accounting Proxy

In section 6.2.1 some notifications are described. This notifications are required by the Accounting Proxy in order to retrieve information that is needed to know

what units have to be monitored, and how to build the SDR documents containing the accounting information. However, the functionality for creating such notifications is not provided directly by the Store. Moreover, this notifications are not required for all the resources registered in the Store that include an URL, so this functionality should not be directly putted in the source code of the Store.

To deal with this notifications, it has been decided to develop a resource plug-in as described in chapter 5. This plug-in defines a type of resource which is being monitored by an Accounting Proxy instance and which is provided as an URL. Following, it is possible to see the *package.json* of the specified plug-in.

```
1  {
2    "defined_type": "Accounting API",
3    "author": "fdelavega",
4    "version": "1.0",
5    "module": "accounting_api.AccountingAPIPlugin",
6    "media_types": [],
7    "formats": ["URL"],
8    "overrides": [],
9    "form": {}
10 }
```

It can be seen, that it has been defined a new resource type called *Accounting API*, which only allows URLs when registering the resource. In addition, given resource type does not limit the allowed media types, nor defines any form, or overrides fields.

Additionally, the plug-in makes use of some of the existing events in order to perform some validations and to send the required notifications:

- **on post create**: This event is being used to validate, using the administration API of the Accounting Proxy, that the user creating the resource is authorized to sell the concrete service.
- **on post binding**: This event is used to send to the Accounting Proxy the binding notification.
- **on post acquire**: This event is used to send to the Accounting Proxy the acquisition notification.

Note that in all the cases described above, the URL of the concrete Accounting Proxy is determined using the URL of the resource provided by the user.

It is necessary to take into account that the created plug-in defines a very general resource type; however, it can be used by FIWARE users, that want to include pay-per-use features in their own solution, as the starting point for the development of their own domain-dependant resource types.

Chapter 7

Revenue Settlement and Sharing

In order to allow FIWARE users that want to make use of the FIWARE Business Framework to monetize composite services and applications where multiple partners are involved, one of the main objectives of the project, described in section 1.3, is supporting complex revenue sharing models able to include multiple stakeholders. In this regard, the current chapter describes the work that have been carried out in order to achieve this objective.

As stated in section 1.1.3, at the beginning if this Master's Thesis project, there were an initial implementation of the RSS, which supported a basic version of the revenue sharing models. Specifically, the RSS only were able to support a revenue sharing calculus based on a fixed percentage distribution, which were described in a simple revenue sharing model that contains the percentage that belongs to one of the involved parts.

It is needed to take into account, that the basic revenue sharing models supported at the beginning of the project only allowed to distribute revenues between two involved parts, the provider of the offering that generate the revenues and the provider of the business infrastructure where the offering were sold.

Moreover, the Store were integrated with this functionality, in order to allow the revenue sharing between its providers and its owners. Concretely, the Store followed a couple of steps when a new RSS instance were registered:

1. The Store created three product classes in the RSS called: single payment, subscription, and pay-per-use.

2. The Store created a simple revenue sharing model for each of the product classes giving the percentage that it want to receive for every one of these classes, specified by the Store admin.

In this way, when the Store charged a customer and generated the CDR document with charging information for the RSS, it included *single payment*, *subscription*, or

pay-per-use as product class, depending on what was the most complex model included in the charged offering.

It can be seen that this approach is very simple, and that limited the possibilities of FIWARE users that want to include revenue sharing features in their solutions, since it does not allow the inclusion of different partners in the business process.

7.1 Revenue Sharing Models

In order to deal the problems existing with simple revenue sharing, it has been redesigned the structure of the different revenue sharing models, in order to contain specific information about the infrastructure provider, digital assets providers, and the applied algorithm.

Concretely, new revenue sharing models must contain the following information:

- **Product Class**: This field identifies the product class which is used to identify the revenue sharing model.
- **Algorithm**: This field is used to identify the algorithm that will be applied for executing the revenue sharing calculus.
- **Aggregator**: This field contains the identification of the aggregator who owns the business infrastructure where the different offerings are being sold. Note that different revenue sharing models can contain different aggregators, which means that a single RSS instance is able to calculate the revenue sharing of multiple charging components (Stores).
- **Aggregator Value**: This field contains the value that will be used to calculate the amount to be paid to the business infrastructure owner during the revenue sharing calculus (e.g the percentage if the algorithm specifies a fixed percentage).
- **Provider**: This field contains the identification of the provider who create the revenue sharing models. Note that this provider is considered as the main owner of the offerings whose revenues are distributed according the current revenue sharing model.
- **Provider Value**: This field specifies the value used to calculate the amount to be paid to the main provider during the revenue sharing calculus.
- **Stakeholders**: This field contains the list of stakeholders that are involved in the revenue sharing. For each stakeholder the following information is required.
  - **Provider**: This field is used to identify a provider who will receive part of the revenue.

- **Provider Value**: This field is used to specify the value used to calculate the amount to be paid to the concrete provider during the revenue sharing calculus.

First of all, the different revenue sharing models must include a product class. This field, as already stated in section 3.3.2, identifies the revenue sharing model, allowing to use the same model in order to distribute the revenue generated by a group of offerings. It is important to remark, that the product class value must be unique for the same provider in a given business infrastructure.

It can be seen that with the new approach, the concrete aggregator is specifically included in the revenue sharing model instead of including it as another provider. This has been done since providers, who are clients of the business infrastructure, typically must not be able to choose the amount they want to share with the platform where they are offering their assets. In this case, aggregators (admins of a Store instance) can provide the value,for the different algorithms available, they require to be included in the revenue sharing models that apply to those assets offered through their infrastructure. When providers want to create a new revenue sharing model they have to provide a product class, an algorithm, the identification of the business infrastructure where they are selling offerings, and the value for them and the rest of stakeholders. In this way, the RSS automatically includes, within the revenue sharing model being created, the value for the aggregator specified by the business infrastructure owner.

In addition, the revenue sharing model specifies the algorithm to be applied to calculate the revenue sharing. In this regard, this field contains an identifier that is interpreted by the RSS in order to determine what method to apply during the revenue sharing calculus. That means that the revenue sharing models do not contains explicitly how to calculate the revenue.

In order to be able to support multiple algorithms the revenue sharing module of the RSS has been modified. Figure 7.1 describes the internal architecture of the revenue sharing module.

In this figure, it is possible to see how the revenue sharing module of the RSS works. In this way, there are thee endpoints that are accessed during the revenue sharing process: (1) the *CDRs Service* is accessed from the Store, which uses it in order to feed charging information contained in CDR documents. This CDRs are then saved in the database through the *CDRs Manager*. (2) The *RS Models service* is used from the Store for the management of the different revenue sharing models, including the creation, update, and deletion of models. Note that this management is actively done by the *RS Models manager* which updates the models in the database, and uses information of providers and aggregators to complete the fields of the model (e.g the aggregator value). (3) The *Settlement Launcher* is used by

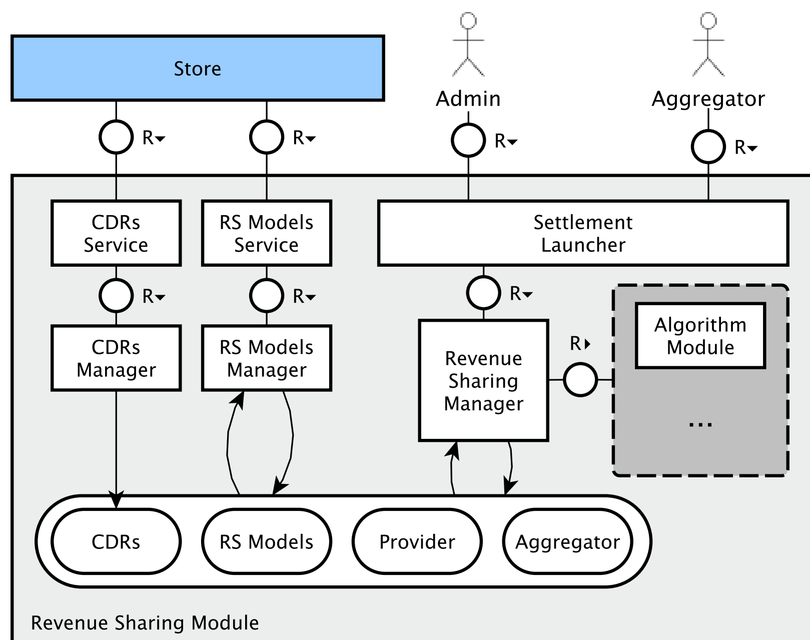


Figure 7.1: RSS Revenue Sharing Module Architecture

admins and aggregators in order to launch the revenue sharing calculus.

It can be seen, that the revenue sharing calculus is performed by the *Revenue Sharing Manager*, which launch the CDR aggregation process as described in section 3.3.2. This module uses the algorithm field contained in the revenue sharing models in order to determine the *Algorithm Module* which is able to calculate the given revenue sharing. These Algorithm Modules implement a common interface that allows both, validating a given revenue sharing model (e.g Fixed percentage module checks whether the values are between 0 and 100, and the aggregation of all the values is not bigger than 100) and executing the revenue sharing calculus for a given model and the aggregation of CDRs for the concrete product class.

Additionally, in order to support the integration of the new revenue sharing models with the offerings model described in chapter 4, it has been crated a RDF vocabulary which allows to describe revenue sharing models in this format. Figure 7.2 show the structure of a revenue sharing model in RDF.

It can be seen that a new vocabulary has been created for representing the information contained in a revenue sharing model. This new vocabulary is identified by the URI *http://rss.fiware.org/sharing* and is included in the diagram using the namespace *rss*. In this approach, the revenue sharing model is described using the class *rss:SharingModel* using the properties *rss:hasAlgorithm* and *rss:hasProductClass* for representing the algorithm and the product class of the model respectively.

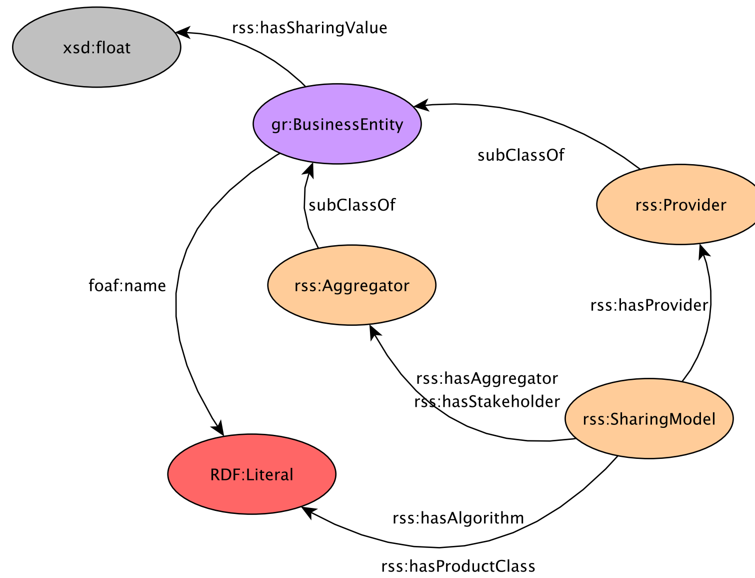


Figure 7.2: RSS RDF Revenue Sharing model

Moreover, aggregators and providers are represented in this model using the classes *rss:Aggregator* and *rss:Provider*, both subclasses of *gr:BusinessEntity*. Additionally, the value used in the revenue sharing models for each of these entities is included using the property *rss:hasSharingValue*. Note that *gr:BusinessEntity* is the class used to represent providers of digital assets in the offerings model, this allows to easily match providers of assets and offerings with the providers included within the RDF version of the revenue sharing models.

In order to match the existing providers and aggregators with the revenue sharing model class, a couple of properties has been defined. The property *rss:hasAggregator* allows to include the aggregator of a revenue sharing model. In addition, the main provider of the model is linked to the *rss:SharingModel* class using the property *rss:hasProvider*. Finally, the rest of stakeholders involved in the revenue sharing are included using the property *rss:hasStakeholder*.

7.2 Integration with the Store

As stated at the beginning of this chapter, the Store was completely integrated with the initial version of the revenue sharing models. In this regard, it has been needed to update the Store in order to support the new version of the revenue sharing models.

First of all, the Store still needs to specify the amount of the revenue, generated by the offerings commercialized in it, it wants to retrieve. To do that, the Store admins (aggregators in the RSS) provide the different aggregator values they want to be included in the different revenue sharing models that specify the concrete Store instance as aggregator (Business infrastructure).

In the new approach, the revenue sharing models are not created automatically by the Store, but for the different offerings providers. It is needed to take into account, that the Store is highly integrated with the RS models APIs provided by the RSS, so it allows providers to retrieve their revenue sharing models or create new ones. In this way, when providers create new offerings in the Store they can provide the product class the offering belongs, so the charging information generated by the new offering can be bound to a given revenue sharing model.

When a customer acquires an offering and she is charged, the Store generates CDR documents containing the charging information required by the RSS. In the new approach, the Store checks the offering information in order to determine the product class it belongs and include it in the different CDR documents, as well as, their own identification as CDR source. This two fields are those that allow to determine the concrete revenue sharing model, and that are used to aggregate CDR documents.

Chapter 8

Smart Searches: The Marketplace and The Repository

The different chapters already covered deal with the enhancing of the FIWARE Business Framework in order to improve features intended to offering providers. However, the FIWARE Business Framework must be also able to support a set of functionalities intended to customers searching for offerings. In this way, section 1.3 states that the FIWARE Business Framework must be able to support smart searches build upon the RDF model described in chapter 4.

At the beginning of the project carried out during the Master's Thesis, the Marketplace and the Repository provide a simple system for searching offerings. In this regard, the Repository stored offerings descriptions as text without interpreting the content. That means that the Repository exposed and API for uploading and downloading complete descriptions.

On the other side, the Marketplace allowed to advertise offerings published in different Store instances by including the URLs of the descriptions of those offerings in the Repository. To support customers searches, the Marketplace downloaded offerings descriptions and created indexes for supporting full text searches.

Despite with the previous approach customers could search for offerings, it had a couple of problems that can make the provided functionality not enough for FIWARE users:

- The offerings model is described in RDF, which means that managing them as text suppose a underutilization of the possibilities of this format. Moreover, using RDF for describing offerings without exploiting its capabilities introduce complexity in the systems without providing any benefit.

- A full text search approach can be enough in simple systems that only require to search using keywords. However, in this case, where offerings from multiple Stores with different classifications and pricing models can be included, it is

required a more flexible mechanism that might allow customers to find the offering that best fits their requirements.

Taking into account the problems existing in the initial version of the Marketplace and the Repository, this chapter covers the solution that have been found in order to achieve the existing objectives.

8.1 Repository RDF Support

In order to be able to exploit the possibilities of having a complete RDF model describing digital assets and offering information, the first component that have been modified is the Repository. In this way, it has been decided to include a new system able to interpret RDF and support SPARQL queries over the stored RDF documents, a triple store.

To deal with this approach, it has been decided to use a concrete system called *Virtuoso* (have a look at section 2.3.2) for the storage of RDF triples belonging to the existing offering and revenue sharing models, since is one of the most extended systems used for this purpose.

It is needed to remark, that the Repository is not only intended to store offering descriptions, but any media file required by the infrastructure where the Repository might be integrated. That means, that not all the resources created within this component will be RDF documents. In this way, it has been decided not to replace the existing storage, but to include the triple store as an extension of the Repository. Figure 8.1 shows the basic architecture of the Repository including the described triple store.

As described in figure 8.1, the Repository saves information about resources and collections as described in section 3.3.4. In this case, the Repository saves the different documents provided in its own storage independently of the format of them. In case the provided document is a RDF document, after saving the meta information of the new resource it the local storage, the *Content Manager* saves the RDF document in the triple store, which parses it and stores the information as RDF triples.

Moreover, the *Content Manager* is also responsible for passing the SPARQL queries made by the existing users, received in the *Query Interface* to the triple store, and builds a response for the users using the information returned by this system.

It can be seen, that with the described approach it has been created a querying engine, which allows other components within the ecosystem to execute smart queries over the different RDF descriptions.

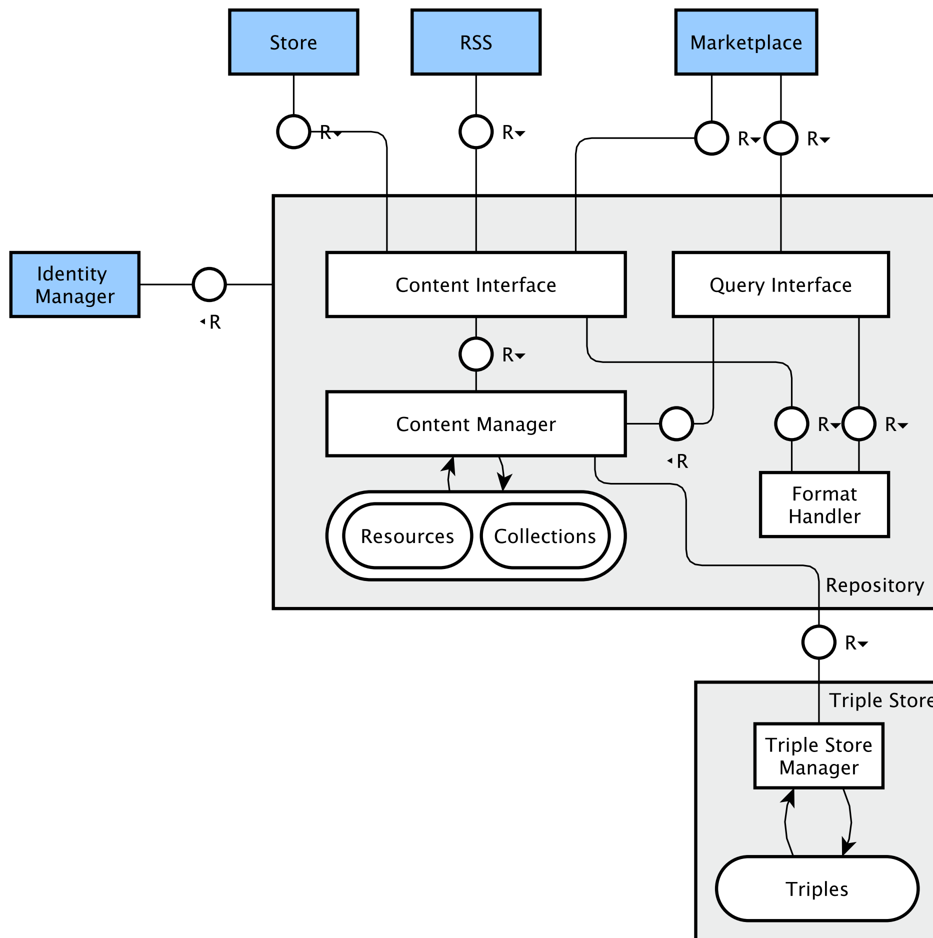


Figure 8.1: Repository Integration with a Triple Store

8.2 Marketplace Support for Searches

Having a querying engine able to support smart queries is required to have a comprehensive searching and discovery system. However, this is not enough for typical customers, that probably will not know how to create SPARQL queries or how to use the querying endpoint. To deal with this problems, the Marketplace has evolved in order to provide some advanced search and discovery capabilities build on top of the querying engine given by the Repository.

In this regard, the Marketplace is completely integrated with the Repository APIs in order to be able to retrieve the different offering descriptions. To do that, offerings are still advertised in the Marketplace by providing the URL of its descriptions in the Repository. However, now the Marketplace parses the existing RDF descriptions and builds its internal structures containing offering information, which can be used by customers in order to visualize the descriptions of the different

offerings.

Although this improvement is quiet simple, it allows customers to view the information of the existing offerings without the need of having to access the Repository and downloading a model that is difficult to read and understand. Note that with this approach, FIWARE users using the Marketplace in their solutions can now use the visualization tools provided in it, instead of having to develop their software for this purpose.

Moreover, the Marketplace also allows customers to perform smart searches. To do that, the Marketplace has two different mechanisms depending on the technical level of the concrete customer. On the hand, for typical customers which do not have technical knowledge, it provides a series of build-in queries that can be configured including some parameters in a visual way, using the provided interface. On the other hand, for customers with knowledge in SPARQL queries, the Marketplace also offers the possibility of directly including the query to be executed in the Repository.

Despite the Marketplace allows to directly include SPARQL queries, this is not the preferred behaviour. It is necessary to take into account that the objective of the Marketplace is allowing FIWARE users to include search and discovery features in their solutions, that probably will be intended to regular users. In this way, the Marketplace focuses its search mechanisms in the build-in queries.

To deal with these build-in queries, the Marketplace offers customers an interface where they can build an configure the query they want to execute. Note that this mechanism has some restrictions, since the Marketplace is intended for the search and discovery of digital assets and offerings; therefore, the results of the queries will be a set of those resources. In this way, customers can choose if they want to retrieve offerings or assets, the type of them, some conditions such as the pricing or legal terms, etc. Following it can be seen some examples of the queries that can be built by Marketplace customers:

- Select all the offerings that contain a given digital asset
- Select all the offerings that contain digital assets of a given category and whose pricing model is based on pay-per-use.
- Select all the digital assets of a given category offered in concrete Store instance.
- Select all the digital assets of a given that are included in multiple offerings under different pricing models.

In order to allow customers to determine the quality of the existing offerings and Stores, the Marketplace supports a review mechanism. In this regard, the Marketplace allows customers to create reviews of offerings and Stores, where they can

include their opinion of them and give a ratting.

Using the reviews, customers searching for offerings not only can customize its searches, but also they can use them to determine whether an offering really is what is being described or if a Store instance is reliable. This gives customers an additional support when having to choose the offering that best fits their needs.

Additionally, the Marketplace offers categorization features of the different digital assets included in an offering. To do that, the Marketplace extracts the category of a given asset using its Linked USDL description retrieved from the Repository. Concretely, it uses the information contained in the class *skos:Concept* included in the RDF description of a Store resource (have a look at chapter 4 for details on this).

Using the information of the type of resource, the Marketplace builds a categorization mechanism which allows customers to browse and search only those offerings that contain assets of the type they are looking for. It is important to remark that this can also be done by using a custom query; however, it has been decided to include this categorization directly in the component in order to easy the searching process and the browsing in the interface.

The categorization feature of the Marketplace also includes recommendations. In this way, when a customer browse the offerings contained in a given category, the Marketplace uses the reviews and ratings of them in order to recommend those with the better rating in order to minimize the time customers spend searching.

Finally, the Marketplace also offers functionality intended for the comparison of offerings. The objective of this feature is supporting the decision of customers when the have found similar offerings. To help customers, the Marketplace allows them to visually put the basic information of offerings as well as its pricing models together in the same interface. This approach helps customers to determine the main differences between the compared offerings, so they can take a decision on what offering acquire.

Chapter 9

Use Case: Energy Consumption Data Market

During this document, it has been described the work that have been carried out in order to enhance the FIWARE Business Framework, and achieve the objectives stated in section 1.3. However, there is also one objective of this project which states that it must be described how the FIWARE Business Framework can be integrated with an existing system in order to allow its owners to include search, discovery, monetization and revenue sharing features. In this regard, the current chapter describes how the FIWARE Business Framework can be integrated in a real environment for allowing the selling of energy consumption data.

9.1 Overview

In the described scenario, there are a couple of meters deployed in some factories that are able to read and publish information on the actual energy consumption. In this regard, the owners of this infrastructure have an Orion Context Broker (see annex B.2 for details in this software) instance deployed, where the different meters upload the energy consumption data periodically. Using this component, the owners of the infrastructure make available the data generated by this meters as real time streams.

For uploading energy consumption information, an entity is created in the Orion Context Broker instance for each meter deployed, and updated periodically every time the meter reads the current consumption value. Following, it can be seen an example of the entity created in the Orion instance for a given meter:

```
{
        "id": "meter6",
        "type": "Meter",
        "isPattern": "false",
        "attributes": [
          {
            "name": "downstreamActivePower",
            "type": "double",
            "value": "1.83"
          },
          {
            "name": "time",
            "type": "timestamp",
            "value": "2015-07-04T09:55:50.997+0200"
          },
          {
            "name": "unitOfMeasurement",
            "type": "string",
            "value": "kW"
          }
        ]
      }
```

It can be seen in the example above that the concrete information belongs to a meter identified as *meter6*. In this case, it contains three attributes that describe the current consumption (*downstreamActivePower*), the timestamp when the value was read, and the unit of measurement, in this case kilo watts.

Using the Orion Context Broker, users are able to subscribe themselves to a concrete meter, so every time the information about the consumption of a given meter is updated they are notified, having this way real time information on the actual consumption of the subscribed meters.

Additionally, owners of the infrastructure also save historical information about the energy consumption during a given period of time as static datasets. To do that, they have a custom software, which is subscribed to the Orion Context Broker and that saves the consumption information in static datasets that it uploads to a CKAN instance (have a look at annex B.3 for details in this software). Specifically, this custom software can be configured specifying the periodicity and duration of the periods it is subscribed to the information of the different meters in order to create the static datasets, which are then saved as csv files named for example *meter6_20150701_20150704.csv*. Following it can be seen an example of the structure of this files.

```
id,     downstreamActivePower, time,                          unit
meter6, 1.83,                  2015-07-04T09:55:50.000+0200, kW
meter6, 1.54,                  2015-07-04T09:56:50.000+0200, kW
meter6, 1.23,                  2015-07-04T09:57:50.000+0200, kW
```

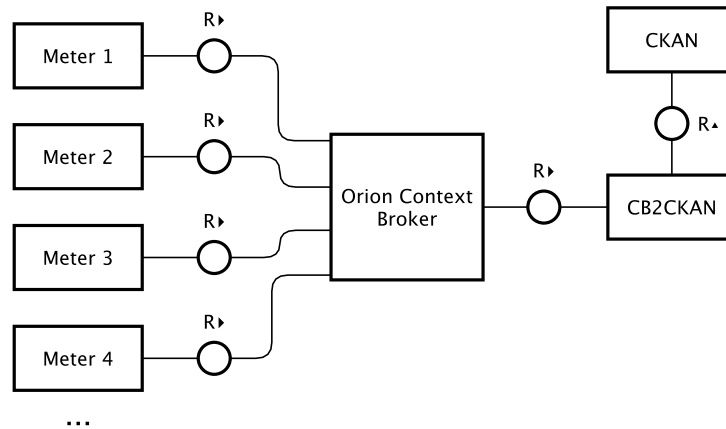


Figure 9.1: Energy Consumption System Architecture

The architecture of the described infrastructure can be found in Figure 9.1. Using this architecture as an starting point, the owners want to monetize this system in order to build an ecosystem able to support the selling of the existing data. Moreover, owners also want to involve external partners in this ecosystem that can provide visualization and analytics tools able to enhance the offered data, making its offerings more attractive to potential customers.

9.2 Data Market Architecture

In order to include monetization, as well as search and discovery features, the existing infrastructure for the retrieving of energy consumption data has been integrated with the different components of the FIWARE Business Framework. In this way, the integration of the data infrastructure and the FIWARE Business Framework build up the called *Energy Consumption Data Market*.

The first step, has consisted in integrating the existing components with an instance of the Identity Manager KeyRock in order to have a unified set of users and organizations able to support the business process and the access control along all the components of the Energy Consumption Data Market. This process has been easily tackle since both Orion Context Broker and CKAN have components made for realizing this task. In the case of the Context Broker it has been done by introducing an instance of the PEP Proxy able to intercept and authenticate requests made to its APIs. On the other hand, in the case of CKAN it has been done by using an existing extension that allows external OAuth2 management of users in this tool.

Since the Energy Consumption Data Market wants to commercialize the existing data, it is required to introduce mechanisms for the authorization of users in order

to control what users have access to the different static datasets and data streams. To do that, the first step has been introducing an instance of the Accounting Proxy covering the Orion Context Broker. In this way, only those users who have acquired the concrete stream will be able to retrieve data from them. Additionally, this approach allows to sell the existing data streams using pay-per-use models, since the Accounting Proxy is in charge of generating usage information (This component is described in detail in chapter 6).

To deal with the access control and accounting functionalities provided by the Accounting Proxy, each of the data streams belonging to a concrete meter has been registered as a monitored service using its URL in the Orion Context Broker as the internal URL, and providing an external path that is used by users in order to access. It is important to remark that in this scenario, the different meters must be able to access the URLs of the Orion Context Broker directly in order to update the current consumption value for the meter. Therefore, in this case it is not possible to hide the URLs of the monitored services just deploying the service in localhost. To deal with this issue it has been followed an approach based on the usage of a firewall that only allows the meters to access the Orion Context Broker without crossing the Accounting Proxy.

In the case of the CKAN instance it is also needed to control what users can access to a given dataset. To do that, it has been used an extension of CKAN that allows to define private datasets that only can be accessed by a chosen group of users.

Once the access control has been covered, the next step was defining the digital assets to be offered. In this regard, its has been defined two types of digital assets: (1) The *Real time Data Stream* describes the real time data offered by a given meter though the Orion Context Broker. (2) The *CKAN Dataset* defines a static dataset which is offered in the CKAN instance. To be able to handle those types of assets in the Energy Consumption Data Market, two plug-ins of the Store defining two types of resources has been created.

On the one hand, it has been created a plug-in for selling real time streams. This plug-in has been developed using the *Accounting API* plug-in described in chapter 6 as the starting point. Following, it can be seen the *package.json* file describing the plug-in:

```
1  {
2    "defined_type": "Real Time Data Stream",
3    "author": "fdelavega",
4    "version": "1.0",
5    "module": "data_stream.RTDataStreamPlugin",
6    "media_types": ["application/json"],
7    "formats": ["URL"],
8    "overrides": [],
9    "form": {}
10 }
```

Example above shows that the new plug-in defines the type *Real Time Data Stream*, which can only be provided by including a URL (The external URL of the monitored service). Moreover, only the *application/json* media type is allowed since this is the media type returned by the Orion Context Broker.

In addition, some event handlers have been implemented for managing the different events related with the real time stream resources:

- **on post create**: This event is being used to validate, using the administration API of the Accounting Proxy, that the user creating the resource is authorized to sell the concrete data stream.
- **on post binding**: This event is used to send to the Accounting Proxy a notification when the concrete data stream resource has been included in a new offering.
- **on post acquire**: This event is used to send to the Accounting Proxy a notification when an offering containing the concrete real time stream has been acquired. This notification is mainly used to authorize new users to access the concrete stream.

On the other hand, it has been created a plug-in for describing CKAN datasets as resources in the Store. Following it is shown the *package.json* file of this plug-in.

```
1  {
2    "defined_type": "CKAN Dataset",
3    "author": "fdelavega",
4    "version": "1.0",
5    "module": "ckan_dataset.CKANDatasetPlugin",
6    "media_types": ["text/csv"],
7    "formats": ["URL"],
8    "overrides": [],
9    "form": {}
10 }
```

In this case, the developed plug-in defines the resource type *CKAN Dataset*, which can only be provided by including the URL of the dataset in the CKAN Instance. In addition, only the text/csv media type is supported for this resource type.

As in the previous case, some event handlers has also been defined for the CKAN dataset resource type. Concretely:

- **on post create**: This event is being used to validate, using the CKAN APIs, that the user creating the resource is authorized to sell the concrete dataset.
- **on post acquire**: This event is used to send to the CKAN instance a notification when an offering containing the concrete dataset has been acquired. This notification is sent to the APIs exposed by the Private Datasets extension of CKAN mentioned above, in order to authorized the customer to access to the concrete dataset.

Finally, the last step for the creation of the Energy Consumption Data Market has been deploying the different components of the FIWARE Business Framework and installing the created plug-ins in the Store instance. Figure 9.2 shows the architecture of the Energy Consumption Data Market.

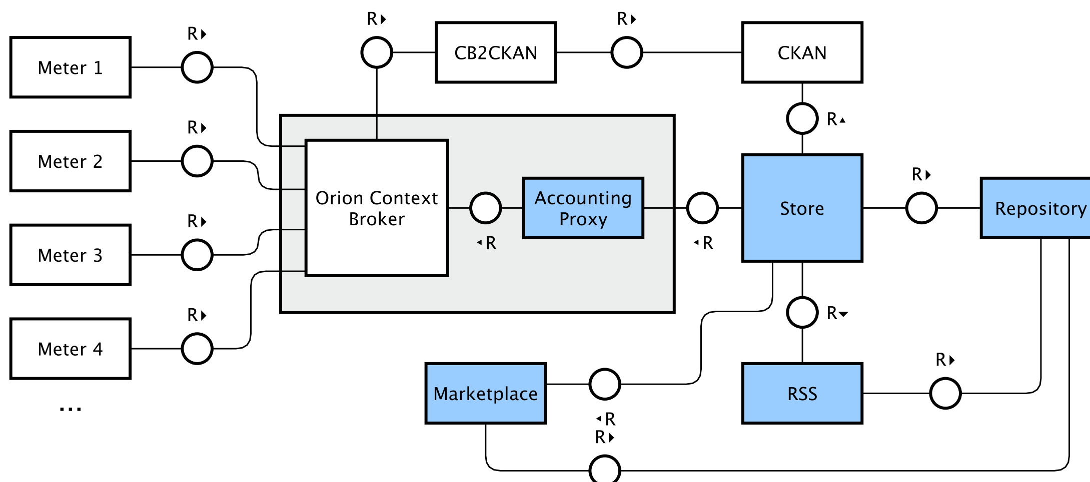


Figure 9.2: Energy Consumption Data Market Architecture

With the architecture described, the created Energy Consumption Data Market allows to create offerings containing both real time streams and datasets, which can then be advertised in the Marketplace in order to allow customers searching for them. Additionally, the integration of the RSS allows that external partners can be involved, by providing additional assets which can also be offered with the existing data and then share the generated revenues.

As examples of the possibilities of the Energy Consumption Data Market some example offerings have been defined, including cases with multiple partners involved. Concretely, it is possible to create offerings such as the following:

- Basic offerings selling each of the real time data streams under pay-pay-use models.
- Basic offerings including the basic datasets for free, used to advertise the existing streams.
- Offerings containing the access to some data streams including a visualization tool provided by an external partner, with a pricing model based on a single payment and pay-per-use. This case also requires a revenue sharing model specifying how to distribute the generated revenues between the data providers and the visualization tool provider.
- Offerings containing premium datasets created by the generation of analytics over the existing data. In this case it is also required a revenue sharing model for distributing revenues between the provider of the premium datasets and the owners of the data used to generate them.

Chapter 10

Results and Conclusions

This chapter describes the results and conclusions that have been extracted during the work that have been carried out in the context of the current Master's Thesis. In this regard, this chapter describes how the given solutions to the existing objectives cover the requirements set by the users of the FIWARE Business Framework (mainly the Use Case Projects and the Accelerators) and what are the main conclusions and comments that can be extracted from these solutions. Additionally, this chapter also describes possible future lines that can be pursued to enhance the FIWARE Business Framework in the future.

Before starting to evaluate the results and conclusions, it is needed to remark the profile of the target users of FIWARE, and thus, the target users of the FIWARE Business Framework. The main objective of the FIWARE Platform is helping service providers and entrepreneurs to create context aware applications by providing innovative technological resources and already implemented APIs. Therefore, target users of the FIWARE Business Framework are those that require enhancing their own solutions with search, discovery, monetization and revenue sharing features.

10.1 Results

It can be seen along the current document, that the objectives proposed in section 1.3 have been tackled one by one in order to enhance the FIWARE Business Framework, with the objective of dealing with the problems identified in the initial version (see section 1.2). It is necessary to remark that the identified problems limited the features offered to potential users, avoiding in some cases that they could make use of the FIWARE Business Framework in their own solutions. As an example, a provider that required pay-per-use models in her application, could not make use of the FIWARE Business Framework if there was not an accounting component already included.

In this regard, chapter 9 shows how the different improvements of the FIWARE

Business Framework, that have been developed as part of the current Master's Thesis, can be applied in a real environment in order to extend existing systems which manage their own digital assets with search, discovery, monetization, and revenue sharing features. This use case uses the whole features provided by the FIWARE Business Framework, so it has been used to evaluate the results of the Master's Thesis project.

The first objective that was taken into account was supporting the monetization of any kind of digital asset. The main problem existing with the monetization of assets was that they were represented using too general models. As has been stated several times during this document, two general types of digital assets were supported: downloadable and APIs, representing assets that were files or URLs respectively. Despite, this way of providing digital assets in the FIWARE Business Framework allowed to include almost any kind of assets, it avoided to perform programmatic validations, check permissions, or in general, any action that require to know the exact format of the type of digital asset.

To deal with the problems related with the monetization of different kind of digital assets, it has been created a system that allows providers to develop extensions of the Store, defining specific types of digital assets. These extensions allow also to create handlers for the existing events within the life cycle of a digital asset within the FIWARE Business Framework (see chapter 5).

The plug-in feature proposed in this Thesis, which allows the development of extensions for the Store, has probed to be one of the most useful enhancements that have been created during the current Master's Thesis project. It can be seen in the chapter 9, that for every type of digital asset being offered (datasets and real time data streams) a new plug-in has been created, allowing the owners of the Data Market to control how their target assets are offered and allowing them to validate the new resources. For example, in the case of the *CKAN Dataset* resource type, created to represent the datasets offered in the CKAN instance, the plug-in validates that the user creating the resource in the Store is the owner of the dataset using the APIs provided by CKAN. This kind of validations were not possible without the plug-in system, so if the created Data Market would had been developed using the initial version of the FIWARE Business Framework, the owners of the Data Market would have needed to modify the source code of the Store. Taking into account the reasoning explained before, it can be said that the plug-ins feature supports the target users needs, since it allows to easily integrate the FIWARE Business Framework with their own solutions.

The next objective that has been evaluated is the support for pay-per-use models, where the main problem was related to the need for accounting. As stated in section 1.2, the initial version of the FIWARE Business Framework supported pay-per-use models; however, it does not performed the accounting of services, delegating this

task in service providers. These approach forced users that wanted to integrate their solution and include pay-per-use assets or just wanted to tack their usage, to develop their own accounting system. Therefore, users needs regarding the integration of a pay-per-use approach, were not completely covered.

To solve the problems regarding the pay-per-use support, it has been developed a new system: the Accounting Proxy, which is able to monitor services using different units such as calls, megabytes, etc (see chapter 6). It can be seen that using the pay-per-use support included in the Store and the functionality provided by the Accounting Proxy, users of the FIWARE Business Framework can integrate complete pay-per-use management in their solutions just by configuring some parameters, without the need of creating ad-hoc components. This is reflected in the chapter 9, where the Accounting Proxy has been deployed to intercept requests to the Orion instance, allowing to offer real time data streams under pay-per-use models and to have information on the their usage. It is important to remark, that the Accounting Proxy is also in charge of authorizing users to access the APIs it is monitoring. Therefore, the inclusion of the Accounting Proxy in the FIWARE Business Framework is giving potential users the possibility of exploiting usage-based business models when monetizing their solutions.

The third objective taken into account has been supporting composite offerings that include digital assets offered by different providers; that is, supporting revenue sharing between multiple providers and stakeholders. The main problem existing in this regard was the lacking maturity of the Revenue Settlement and Sharing System, which only supported simple revenue sharing models between one provider and the owner of the infrastructure where the offerings were being sold. This represented a lack of functionality for FIWARE users interested in integrating the Business Framework with their solutions, since it limited the complexity of the offerings being sold and the number of stakeholders that could be involved.

The problems with revenue sharing models have been tackled by updating the core of the Revenue Sharing System in order to support more complex and flexible revenue sharing models, which can include multiple stakeholders, as stated in chapter 7. As can be seen in chapter 9, the new revenue sharing models are flexible enough to allow the distribution of revenues in a real environment. In the case of the Data Market, it has been created some offerings that include in a single bundle datasets, real time data streams, and visualization tools, provided by different providers. In this case, it was required to create a revenue sharing model which specifies how to distribute the revenue generated by those offerings between the dataset owner, the Orion owner, the visualization tool provider, and the business infrastructure owner. In conclusion, the new revenue sharing models give users of the FIWARE Business Framework the possibility of integrating more complex offerings, and extending their business model to include more partners or stakeholders.

Finally, it has been evaluated the objectives regarding search, discovery and comparison of offerings, as well as the RDF based model for describing them, used as the basis for supporting the stated features. As described in section 1.2, there were a couple of problems regarding these objectives. First, despite the initial version of the FIWARE Business Framework already used a RDF model for describing offerings (Linked USDL documents), those documents only contained part of the information of the related offerings. Moreover, there were not a triple store or a similar system able give real support for making queries over the Linked USDL documents of the offerings, so the possibilities of having this kind of model were not exploited. Finally, taking into account that there were not any querying engine, the Marketplace only were able to support full text searches over the offering descriptions.

It can be seen, that the needs of the FIWARE Business Framework users regarding the integration of search and discovery features were not sufficiently covered, since the initial versions of the Repository, the Marketplace and the Linked USDL documents only allowed to perform simple searches.

To deal with the problems related to the search, discovery and comparison of offerings some features have been developed. First, as can be seen in chapter 4, the Linked USDL model for describing offerings has been evolved to include all the needed information such as the description of the digital assets being sold. Then, both the Repository and the Marketplace have been updated (see chapter 8). In this regard, a triple store (Virtuoso) has been integrated with the Repository and new APIs for searching have been created, allowing to execute SPARQL queries over offering descriptions. On the other hand, the Marketplace has been modified for including support to the new searching features provided by the Repository and for comparison of offerings based on the types of assets being sold and its pricing models. Taking into account the new search and discovery features that have been integrated in the FIWARE Business Framework, FIWARE users can integrate it with their own solutions in order to easily include smart searches to their monetization services.

10.2 Conclusions

During the development of the objectives planned for this Master's Thesis, it has been possible to extract some conclusions and comments, as well as some lessons learned.

First of all, it has been possible to determine that a general solution is not always the best option, even if it is more flexible. This is very clear when taking about the digital assets that are offered in the FIWARE Business Framework, where the existing model in the initial version allowed to include almost any kind of digital asset, but on the other side, it did not allow to perform some actions that require to know the format of the resource, so the framework did not provide a real solution to potential users.

In addition, it has been learned that in complex systems such as the FIWARE Business Framework, composed of independent components intended to work highly integrated, a way of having a common user management between all the components becomes crucial in a real scenario. Similarly, an integrated way of managing the common information such as the offering model allows to avoid dealing with different formats, easing the integration process.

Regarding the results obtained when integrating the FIWARE Business Framework in a real environment, it can be said, that the provided functionality has been significantly improved since the initial version, going from a system that forced users, who wanted to include the framework in their solutions, to develop a couple of support components (e.g an accounting system) or event to modify the source code of the existing components, to a system which provides users complete support in search, discovery, monetization and revenue sharing features. Therefore, it can be seen that the enhancing of the FIWARE Business Framework that have been carried out, has given FIWARE users the possibility of easily including business features in their applications without too many development.

Finally, as a success story of the work carried out during this Master's Thesis, the journal *Novatica* [20] has proposed to include an article in their special number, published for their 40th anniversary, regarding the business ecosystem and the FIWARE Business Framework enhanced in this project.

10.3 Future lines

The development of the FIWARE Business Framework has been carried out in the context of the FICORE project. In this regard, it is important to notice that all the work done in the context of this Master's Thesis does not finished with it, but it is still going to evolve during the following months or years. Thus, a couple of future lines have been identified for its short-term roadmap.

First, it is planned to extend the existing pricing models in order to include a new basic one based on revenue sharing. The objective of having this kind of models, is allowing providers to be able to create price plans that allowed developers to acquire offerings without explicitly paying for them. However, those developers will be forced to share part of the revenues generated by the usage of the acquired offerings in their own offerings.

Taking into account that during the usage of the FIWARE Business Framework both accounting and charging information is generated, it has been decided to integrate the framework with reporting tools (e.g Pentaho) in order to allow the creation of reports and analytics that can be used as the stating point of a business intelligence process.

The current architecture supports the monetization of services, by registering them as resources using their endpoint. In general, this means that if the service needs to be moved to a different location, then a new version of the resource needs to be provided. To give more flexibility to the FIWARE Business Framework, a new component, called the Registry, will be introduced. The objective of this component is to provide a functionality similar to UDDI, by maintaining runtime information of the services registered in the framework, including their endpoints and some configuration parameters.

Additionally, it has been proposed to develop more accounting modules for the Accounting Proxy (see section 6.2.1 for details on what is an accounting module) in order to make available to providers more pay-per-use units.

Regarding the Marketplace, it is wanted to develop a component able to give offering recommendations to the existing customers based on its user's profile and context in comparison to explicit semantics of available offerings as well as previous activities and experiences of the marketplace participants (Wisdom of the crowd and social networks).

In addition, it has also been proposed to allow owners of a Marketplace instance to include domain specific configuration and ontologies. This will allow FIWARE users to include within their solution domain specific searching and comparison features. For example, for creating a hotels searcher that allows to compare hotels based on the number of starts.

For the RSS it is needed to develop new Algorithms modules (have a look at chapter 7 for details on what is this component), able to support more algorithms in the revenue sharing calculus.

Finally, some FIWARE users have shown interest in using the FIWARE Business Framework, not only for monetizing digital assets, but also real products and physical services. In this regard, it is necessary to remark that the framework does not provide some features that are required for this kind of systems such as tacking of the product, maintaining the available stock, managing refunds, etc. Therefore, it will be analysed the possibility of developing an extension of the Store with all the missing features in order to attend the needs demanded by FWARE users.

Appendix A

Fundamental Modeling Concepts Format

Fundamental Modeling Concepts (FMC) [12] defines a consistent and coherent format to represent dynamic systems. It enables to communicate concepts and structures of complex informational systems in an efficient way among the different types of stakeholders. A universal notation originating from existing standards, easy to learn and to apply, is defined to visualize the structures and to communicate in a coherent way. In contrast to most of the visualization and modeling standards of today it focuses on human comprehension of complex systems on all levels of abstraction by clearly separating conceptual structures from implementation structures. FMC is based on strong theoretical foundations, has successfully been applied to real-life systems in practice (at SAP, Siemens, Alcatel etc.) and also is being taught in software engineering courses at the Hasso Plattner Institute for Software Systems Engineering.

FMC Block diagrams show the compositional structures as a composition of collaborating system components. In this format, there are active systems components called agents, and passive system components called locations. Each agent processes information an thus serves a well-defined purpose. Therefore, an agent stores information in storages and communicates via channels or shared storages with other agents.

Basic Elements

Figure A.1 shows the active components in FMC, an agent and a human agent, that serves a well defined purpose and therefore has access to adjacent passive systems.

Figure A.1: FMC Active Componets

Figure A.2 shows the passive components in FMC, an storage and a communication channel. A storage is used by agents for storing data, and a channel is used for communication purposes by at least two active system components.

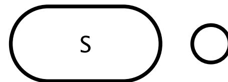


Figure A.2: FMC Passive Componets

Common Structures

Figure A.3 shows a basic read access where agent A reads from storage S.

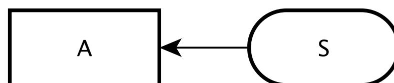


Figure A.3: FMC Read Access

Figure A.4 shows a basic write access where agent A writes to storage S.

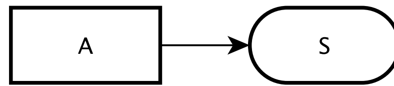


Figure A.4: FMC Write Access

Figure A.5 shows a read-write access (Modification) where the agent A is able to modify the storage S.

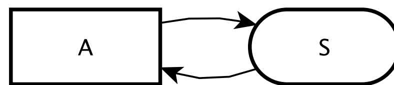


Figure A.5: FMC Read-Write Access

Figure A.6 shows a request/response communication channel where agent A1 request information to agent A2 which in turn responds (e.g functions or HTTP channels).

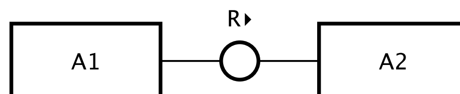


Figure A.6: FMC Request - Response

Appendix B

Third Party Components

B.1 Identity Manager GE: KeyRock

Identity Management covers a number of aspects involving users' access to networks, services and applications, including secure and private authentication from users to devices, networks and services, authorization and trust management, user profile management, privacy-preserving disposition of personal data, Single Sign-On (SSO) to service domains and Identity Federation towards applications. The Identity Manager is the central component that provides a bridge between IdM systems at connectivity-level and application-level. Furthermore, Identity Management is used for authorising foreign services to access personal data stored in a secure environment. Hereby usually the owner of the data must give consent to access the data; the consent-giving procedure also implies certain user authentication.

Identity Management is key on any architecture. IdM offers tools for administrators to support the handling of user life-cycle functions. It reduces the effort for account creation and management, as it supports the enforcement of policies and procedures for user registration, user profile management and the modification of user accounts. Administrators can quickly configure customized pages for the inclusion of different authentication providers, registration of tenant applications with access to user profile data and the handling of error notifications. For end users, the IdM provides a convenient solution for registering with applications since it gives them a means to re-use attributes like address, email or others, thus allowing an easy and convenient management of profile information. Users and administrators can rely on standardised solutions to allow user self-service features. In addition to providing a native login, IdM supports the integration of multiple 3rd party authentication providers. Foremost, it supports in a first step the configuration of preferred identity providers through the administrators. The use of 3rd party IdMs lowers the entry barriers for a native user to register, since the user can link to her/his preferred IdM and use this account for authentication. As it is possible to configure several applications that shall be linked to his IdM, the main benefit for users is a single sign-on (SSO) to all these applications. The IdM offers hosted user

profile storage with specific user profile attributes. Applications do not have to run and manage their own persistent user data storages, but instead, can use the IdM user profile storage as a Software as a Service (SaaS) offering.

The KeyRock Identity Management GEI complies with existing standards for user authentication and it provides access information to services acting as a Single Sign-On platform. The KeyRock IdM is a free/open source software which can be integrated with any development, specially with any Cloud service.

KeyRock provides a set of interfaces that can be used in order to authenticate users and provide single-sign-on features using different protocols. In this regard, KeyRock offers an OAuth2 interface which can be used by the different components of the FIWARE Platform, for authenticating requests and for obtaining the access tokens required for accessing protected resources. Therefore, KeyRock provides the Authentication Server functionalities as described in the OAuth2 protocol specification [21].

Additionally, with KeyRock it is also offered an external component intended to support authenticated access of those applications registered as protected resources: the PEP Proxy. This component is a basic software which intercepts the different requests made to the concrete protected resource, and validates them. To do that, the PEP Proxy extracts the access token that must be included in the requests and validates it using KeyRock. This way, users are not only offered a system which can be used for authenticating users and authorizing accesses, but a also a component that can be used to actually protect their services and resources.

B.2 Context Broker GE: Orion

The Context Broker GE enables publication of context information by entities, referred as Context Producers, so that published context information becomes available to other entities, referred as Context Consumers, which are interested in processing the published context information. Applications or even other GEs in the FIWARE platform may play the role of Context Producers, Context Consumers or both. Events in FIWARE based systems refer to something that has happened, or is contemplated as having happened. Changes in context information are therefore considered as events that can be handled by applications or FIWARE GEs.

The Context Broker GE supports two ways of communications: push and pull towards both the Context Producer and the Context Consumer. It does mean that a Context Producer with a minimal or very simple logic may continuously push the context information into the Context Broker, when the information is available or due to the internal logic of the Context Producer. The Context Broker on its side can request the context information from Context Producers if they provide

the ability to be queried (Context Producers able to act as servers are also referred as Context Providers). In a similar way, Context Consumers can pull the context information from the Context Broker (on-request mode), while the Context Broker can push the information to Contest Consumer interested in it (subscription mode).

A fundamental principle supported by the Context Broker GE is that of achieving a total decoupling between Context Producers and Context Consumers. On one hand, this means that Context Producers publish data without knowing which, where and when Context Consumers will consume published data; therefore they do not need to be connected to them. On the other hand, Context Consumers consume context information of their interest, without this meaning they know which Context Producer has published a particular event: they are just interested in the event itself but not in who generated it. As a result, the Context Broker GE is an excellent bridge enabling external applications to manage events related to the Internet of the Things (IoT) in a simpler way hiding the complexity of gathering measures from IoT resources (sensors) that might be distributed or involving access through multiple low-level communication protocols

In this regard,the Orion Context Broker is an implementation of the Context Broker GE, providing the NGSI9 and NGSI10 interfaces. Using these interfaces, clients can do several operations:

- Register context producer applications, e.g. a temperature sensor within a room
- Update context information, e.g. send updates of temperature
- Being notified when changes on context information take place (e.g. the temperature has changed) or with a given frequency (e.g. get the temperature each minute)
- Query context information. The Orion Context Broker stores context information updated from applications, so queries are resolved based on that information.

B.3 CKAN

CKAN is a powerful data management system that makes data accessible, by providing tools to streamline publishing, sharing, finding and using data. CKAN is aimed at data publishers (national and regional governments, companies and organizations) wanting to make their data open and available. In this way, CKAN is a fully-featured, mature, open source data management solution. CKAN provides a streamlined way to make the different data discoverable and presentable. Each dataset is given its own page with a rich collection of metadata, making it a valuable

and easily searchable resource.

CKAN allows to publish and manage data by providing an intuitive web interface which allows dataset publishers and curators to easily register, update and refine datasets in a distributed authorisation model called *Organizations*. *Organizations* allow each publisher to have their own dataset entry and approval process with numerous members. This means responsibility can be distributed and authorization access managed by each department or agencies' admins instead of centrally.

Many organizations already have their data in repositories with well-defined process and procedures for publishing and managing data. In this case the data can be simply pulled regularly into CKAN from the existing repositories. To facilitate this model CKAN provides a sophisticated and customisable *harvesting* mechanism which can fetch and import records from many different repository sources.

CKAN provides a rich search experience which allows for quick Google-style keyword search as well as faceting by tags and browsing between related datasets. Users can quickly see what datasets are available, in which formats and with which licence, straight from the search results. In is needed to remark that all dataset fields are searchable.

References

[1] Amazon AWS. https://aws.amazon.com/marketplace. Accessed: 05/07/2015.

[2] App Stores Statistics. http://www.statista.com/topics/1729/app-stores/. Accessed: 06/05/2015.

[3] Apple App Store (Requires iTunes). http://www.itunes.com/appstore/. Accessed: 06/05/2015.

[4] Blackberry World. https://appworld.blackberry.com/webstore/. Accessed: 05/07/2015.

[5] CloudStore. https://www.gov.uk/digital-marketplace. Accessed: 05/07/2015.

[6] DCTERMS. http://dublincore.org/documents/dcmi-terms/. Accessed: 11/06/2015.

[7] FI-PPP. http://www.fi-ppp.org/. Accessed: 05/05/2015.

[8] FIWARE Academy. http://edu.fiware.org/. Accessed: 05/05/2015.

[9] FIWARE Catalogue. http://catalogue.fiware.org/. Accessed: 05/05/2015.

[10] FIWARE Lab. http://lab.fiware.org/. Accessed: 05/05/2015.

[11] FIWARE Platform main page. http://www.fiware.org/. Accessed: 04/05/2015.

[12] FMC Home page. http://www.fmc-modeling.org/. Accessed: 15/06/2015.

[13] FOAF Home page. http://xmlns.com/foaf/spec/. Accessed: 28/06/2015.

[14] Fujitsu Cloud Store. http://www.fujitsu.com/fts/solutions/cloud/. Accessed: 05/07/2015.

[15] GetJar. http://www.getjar.com. Accessed: 05/07/2015.

[16] GoodRelations Home page. http://www.heppnetz.de/ontologies/goodrelations/. Accessed: 28/06/2015.

[17] Google Play. http://play.google.com/. Accessed: 06/05/2015.

[18] LG Smart World. http://www.lgappstv.com. Accessed: 05/07/2015.

[19] Linked USDL. http://www.linked-usdl.org/. Accessed: 05/05/2015.

[20] Novatica Home page. http://www.ati.es/novatica/. Accessed: 03/07/2015.

[21] OAuth2 Documentation. http://tools.ietf.org/html/rfc6749. Accessed: 15/06/2015.

[22] OpenLink Virtuoso Home Page. http://virtuoso.openlinksw.com/. Accessed: 05/07/2015.

[23] Opera Mobile Store. http://ovi.sigma.apps.opera.com/. Accessed: 05/07/2015.

[24] Orange App Shop. http://www.orangepartner.com/articles/distribution-channel-app-shop#.UZFDpivOQ7Q. Accessed: 05/07/2015.

[25] PAV Home page. http://pav-ontology.github.io/pav/. Accessed: 28/06/2015.

[26] RDF W3C. http://www.w3.org/RDF. Accessed: 05/05/2015.

[27] Samsung App Store. http://www.samsung.com/es/apps/mobile/. Accessed: 05/07/2015.

[28] SKOS Home page. http://www.w3.org/2004/02/skos/. Accessed: 28/06/2015.

[29] SOAP W3C. http://www.w3.org/TR/soap. Accessed: 06/05/2015.

[30] Sprint Digita Lounge (Passed Away page). https://manage.sprintpcs.com/digitallounge/home. Accessed: 05/07/2015.

[31] Verizon Wireless. http://www.verizonwireless.com/. Accessed: 05/07/2015.

[32] WADL W3C. http://www.w3.org/Submission/wadl/. Accessed: 06/05/2015.

[33] White Sky's "My Cloud Store". http://www.mycloudstore.co.nz/. Accessed: 06/05/2015.

[34] Windows Phone Market. http://www.windowsphone.com/en-us/store. Accessed: 05/07/2015.

[35] WS Agreement. https://www.ogf.org/documents/GFD.107.pdf. Accessed: 06/05/2015.

[36] WSDL W3C. http://www.w3.org/TR/wsdl. Accessed: 06/05/2015.

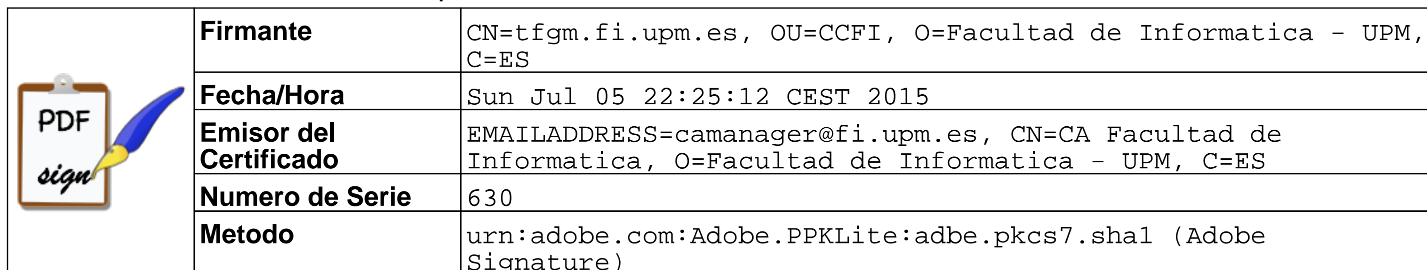
[37] Jones C. Apps with in-app purchase generate the highest revenue, forbes, march 2013. http://www.forbes.com/sites/chuckjones/2013/03/31/apps-with-in-app-purchase-generate-the-highest-revenue/. Accessed: 05/07/2015.

[38] J. Meulen R. Van der; Rivera. Forecast overview: Public cloud services, worldwide, 2011-2016, 4q12 update, gartner research report, gartner inc. 2013. http://www.gartner.com/resId=2332215. Accessed: 06/05/2015.

[39] B.; Müller, R. M.; Kijl and K. J Martens. A comparison of inter-organizational business models of mobile app stores: There is more than open vs. closed. in journal of theoretical and applied electronic commerce research. vol. 6, nº 1, pp. 63-76, august 2011.

[40] Finley I.; Redman P.; Prentice B.; Buchanan S. Enterprise app stores can increase the roi of the app portfolio. gartner report, february 2013. http://www.gartner.com/resId=2325115. Accessed: 06/05/2015.

Este documento esta firmado por



	Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Sun Jul 05 22:25:12 CEST 2015
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)